MASTER EN SCIENCES INFORMATIQUES

Étude d'un générateur de rapports sur bases de données IMS

Guebels, Marc; Lanssens, Philippe

Award date:
1984

Awarding institution:
Universite de Namur

Facultés Universitaires Notre Dame de la Paix, Namur

Institut d'Informatique

Année académique 1983-1984

ETUDE D'UN GENERATEUR DE RAPPORTS SUR BASES DE DONNEES IMS

Marc GUEBELS
Philippe LANSSENS

Mémoire présenté en vue de l'obtention du grade de liciencié et maître en informatique.

Nous tenons à remercier Monsieur Jean-Luc HAINAUT, notre professeur, d'avoir accepté de diriger ce mémoire.

Nous exprimons toute notre gratitude à Monsieur Marcel WAGNER, Directeur du Personnel, à Monsieur Jean-Claude LECOMTE, attaché au comité de Direction et à Monsieur Yves BRASSEUR, ingénieur principal de nous avoir autorisés à effectuer notre stage au service informatique de l'ARBED.

Nous adressons notre vive reconnaissance à Messieurs René ECKER, ingénieur et Thierry MUSCHANG, attaché scientifique pour leurs conseils qui nous ont permis de progresser dans ce travail.

Notre reconnaissance s'adresse aussi à tous les membres du service informatique d'ARBED Luxembourg pour leur bon accueil et tout particulièrement à Mademoiselle Monique RICKAL qui a réalisé la mise en page et la dactylographie de ce mémoire, ainsi qu'à toutes les personnes qui, d'une manière ou d'une autre, nous ont aidés au cours de cette année.

TABLE DES MATIERES.

INTRODUCTION

Il est sans doute superflu d'évoquer l'essor qu'a pris, ces dernières années, l'emploi des bases de données, tant dans les entreprises que dans les pouvoirs publics.

Une base de données centralise des informations destinées à être traitées. Un de ces traitements peut consister à les utiliser à des fins de rapports sur l'activité d'une société. D'autre part, les ressources en personnel informatique se raréfiant, la communauté informatique tend à penser à l'emploi de générateurs automatiques de ces rapports.

Aussi, ce mémoire étudie la possibilité d'une telle implémentation. La démarche suivie pour rédiger ce mémoire est illustrée par la figure 0.1. Des problèmes généraux à l'informatique, ainsi que les problèmes et contraintes propres à l'ARBED nous ont permis d'établir des spécifications de l'outil demandé. Au vu de celles-ci, ainsi que de produits existants et de modèles développés à l'Institut d'Informatique à Namur, nous avons établi l'architecture d'un système répondant à ce cahier de charge. Cette architecture se décompose en des interfaces et des composants.

Cette méthode nous a conduits à découper le mémoire en les chapitres suivants. Dans le premier, nous traitons des problèmes actuels de l'informatique, ainsi que de ses tendances. Dans la deuxième partie de ce chapitre, nous dressons un bref aperçu de l'ARBED, de son service informatique et d'un projet d'organisation future.

Les deux chapitres suivants exposent des concepts qui sont à la base du système. Le Modèle d'Accès Généralisé décrit les structures de données et les caractéristiques d'accès à ces données. Le dictionnaire de données répertorie les informations contenues dans les bases de données.

Le chapitre quatre est consacré aux spécifications auxquelles le système doit répondre et de l'environnement dans lequel il se trouvera.

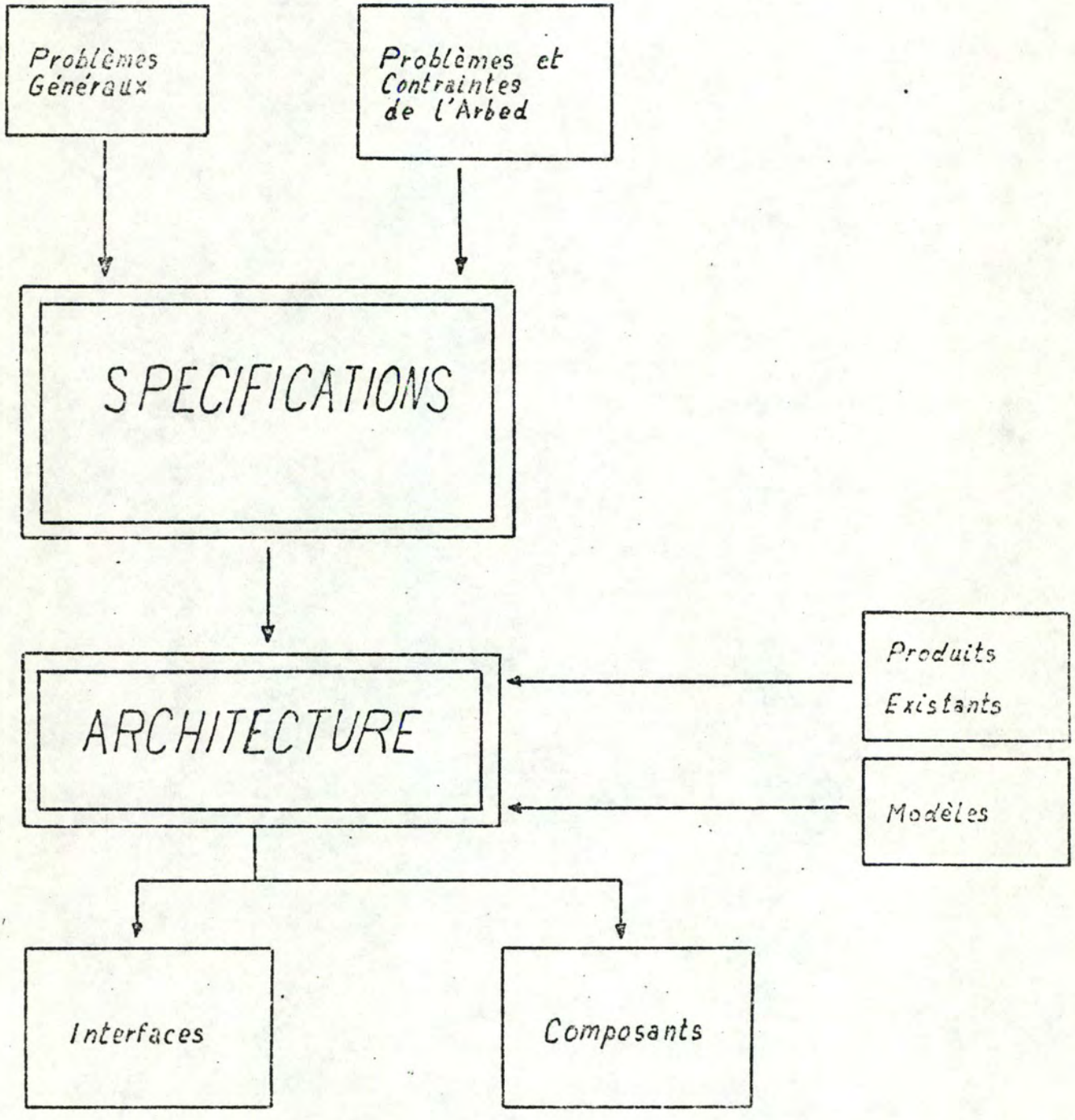


Fig. 0.1 : La Charpente du Mémoire.

Le chapitre cinq est dédié à une présentation de l'architecture et constitue donc un préambule aux autres chapitres que nous consacrons à une explicitation des différents composants du système.

Pour entrer sa requête, l'utilisateur doit définir son rapport selon deux langages. Ceux-ci constituent des interfaces entre l'homme et le système, et représentent des descriptions logique et algorithmique du rapport. La description de ces deux langages constituent les chapitres six et sept.

Le système est développé dans un certain environnement qui se caractérise par un système de gestion de bases de données (IMS) et un dictionnaire de données (DADIC). Tous deux contraignent le système. Aussi, nous leur consacrons un chapitre à chacun.

Quand l'utilisateur a introduit sa définition de rapport, celle-ci est analysée systématiquement afin de détecter des erreurs le plus rapidement possible. Si aucune erreur n'est détectée, l'analyse aura établi une représentation interne du rapport, plus adéquate aux traitements ultérieurs. La présentation de ces contrôles et de la représentation interne constitue un chapitre important.

L'analyse serait bien inutile, si son résultat ne servait à générer un code compilable et exécutable. Un chapitre est consacré à développer comment aboutir à ce code.

Avant de conclure en évoquant des perspectives que nous espérons de notre étude, nous évoquons les grands principes qui doivent gérer le concept utilisé dans la définition logique du rapport.

Chapitre 1: Le contexte du mémoire

1.1. L'évolution de l'informatique

1.1.1. Le dilemme de la programmation

Dans la plupart des entreprises, la demande de nouvelles applications dépasse ce que peut fournir le service informatique. Ce déséquilibre entre la demande et l'offre s'accroît en permanence. Cette situation ne peut qu'empirer avec la chute continue des prix du matériel, car cette dernière autorise une plus grande informatisation de la société... Ce phénomène est frustrant pour les utilisateurs finals qui, non seulement attendent trop longtemps la réalisation de leurs applications, mais surtout n'osent plus envisager ni demander de nouvelles réponses à leurs besoins.

L'évolution de la demande de programmation est supérieure à la croissance du nombre de programmeurs potentiels. La formation ne suivant pas, il faut apporter des changements à la méthodologie et à l'organisation de l'informatique. Le tendance actuelle serait de trouver le juste milieu entre:

- détourner une partie de la charge de développement, de programmation sur l'utilisateur lui-même. Les services informatiques des entreprises doivent apprendre à se procurer sans programmeurs les applications dont ils ont besoin chaque fois que c'est possible. Il s'agit de mettre entre les mains de l'utilisateur des outils lui permettant de créer ses propres applications. Il existe de nombreux exemples où l'utilisateur final participe aux travaux informatiques et obtient de biens meilleurs résultats que de vrais informaticiens, simplement parce qu'il connaît les subtilités de la tâche à réaliser.

- et augmenter la productivité du programmeur. Toutes les applications ne sont pas susceptibles d'être résolues par les nouveaux outils. La programmation qui revient au service informatique doit se réaliser à un rendement supérieur à celui d'aujourd'hui, par des méthodes nouvelles car l'usage généralisé de la programmation structurée ne suffit plus.

1.1.2. De nouveaux outils

Faire participer l'utilisateur final à l'informatique, constitue un changement radical. Ceci exige un logiciel d'un abord facile et donc amical pour l'utilisateur, des dialogues d'applications et la formation d'un grand nombre d'utilisateurs. Cela demande à l'analyste système de jouer le rôle de guide, de professeur pour les utilisateurs en leur montrant les possibilités des nouvelles méthodes.

Les techniques traditionnelles de l'analyse système prennent beaucoup de temps. Le besoin vital de productivité implique l'accélération de l'analyse système autant qu'il nécessite d'accélérer la programmation. Les outils de haut niveau éliminent le besoin d'organigrammes et souvent de circulation de données. Ils sont souvent auto-documentés et évitent le besoin d'écrire les spécifications du programme, mais cette documentation ne suffit pourtant pas toujours à l'utilisateur non averti.

Sept catégories de logiciel permettent aux analystes systèmes de créer des applications:

- des outils d'interrogation simple, permettant d'imprimer ou d'afficher les fichiers enregistrés dans un format convenable.

- des langages d'interrogation complexe. Pour l'utilisateur de bases de données, il existe des langages permettant la formulation d'interrogations portant sur de multiples fichiers.

Parfois l'interrogation fait appel à la recherche d'une base de données complexe ou à l'union d'enregistrements multiples.

- des extracteurs-éditeurs généralisés (générateurs de rapports). Ceux-ci permettent la mise en forme de résultats complexes avec des possibilités assez grandes de traitements (logiques et algébriques) associées aux fonctions d'édition. Ces outils nécessitent une formalisation préalable du besoin, pouvant donner lieu à des "maquettes" de sortie que l'on met au point avec l'utilisateur final.

  La recherche d'information nécessite souvent l'extraction d'enregistrements multiples dans les fichiers (ou BD); sélections, tris sont également possibles et intégrés au langage d'édition. Selon les produits, le processus de mise en forme est plus ou moins automatisé ou guidé par des instructions détaillées. L'exécution peut être faite en mode "batch" et/ou "on-line" par génération intermé- diaire de programmes compilables ou par interprétation et exécution directes.

- des langages graphiques. Des fonctions simples de présentation attractive de résultats sous forme graphique peuvent être rattachées aux éditeurs généralisés et/ou aux outils d'aide à la décision.
  Mais au-delà des fonctions de sortie graphique, il existe des langages interactifs pour lesquels les facilités graphiques constituent une véritable aide à la conception des applications ou des objets du système d'information de l'utilisateur.

  Aux fonctions de pures mathématiques graphiques sont intégrées des fonctions de traitement (logique et algébrique) ainsi que des fonctions de gestion de bases de données particulières.

- des générateurs d'application. Ils contiennent des modules qui permettent de produire une application entière. L'on peut spécifier l'entrée, sa validation, quelle action elle va provoquer, les détails de l'action, les opérations arithmétiques et logiques et quelle sortie est créée. La plupart des générateurs d'application fonctionnent avec des bases de données.

- des langages de programmation de très haut niveau. Certains (ADA, ...) intègrent tous les nouveaux concepts de la théorie des langages déterministes et de l'algorithmique. Ils sont plus puissants et plus compacts que les langages classiques. D'autres (Prolog, ...) sont axés sur la représentation des expressions symboliques et le traitement d'objets formels (listes, arbres, etc.) de façon déterministe.

- des progiciels paramétrés. On peut acheter des progiciels pour certaines applications. Ils exigent souvent énormément de travail sur mesure et d'adaptation pour s'intégrer dans l'organisation qui les installe et sont conçus avec des paramètres qui peuvent être choisis pour modifier leur fonctionnement. Il faut une documentation claire, digestible par l'utilisateur, écrite dans la langue de l'entreprise et qui n'introduise donc pas une nouvelle terminologie. Ces progiciels interdisent généralement les corrections de programmes et l'entreprise doit adapter son organisation à celle du package.

Quant à savoir si ces outils conviennent à l'utilisateur, il faut considérer cas par cas. On peut dire qu'un programme convient pour l'utilisateur final, si ce dernier peut apprendre à l'utiliser et obtenir de bons résultats après deux jours de formation et ne pas l'oublier s'il l'abandonne pendant plusieurs semaines. Pour qu'un langage soit intéressant, il faut pouvoir commencer à l'utiliser très facilement, mais que l'utilisateur puisse continuer à se perfectionner et à s'améliorer pendant longtemps le poussant ainsi à franchir le seuil qui sépare les profanes des initiés du club...

Ce jeu de langages est appelé parfois langage de quatrième génération. Le langage machine formait la première génération, les langages du niveau assembleur la seconde, et les langages indépendants de la machine (COBOL, PLI, BASIC, ...) constituaient la troisième.

1.1.3. Des spécifications plus rigoureuses

A lui seul, l'accroissement de la productivité serait une puissante raison de passer de la programmation classique à des formes plus automatisées. Une autre raison se révèle souvent encore plus impérieuse: dans de nombreux cas, le développement classique ne marche pas.

En d'autres termes, les utilisateurs prétendent que le système développé ne représente pas ce qu'ils désirent. Ou alors, ils se rendent compte après avoir utilisé le système pendant quelques semaines, qu'ils désirent quelque chose de différent.

Une réaction à cette situation consiste à dire que les exigences ne sont pas spécifiées avec suffisamment de précision ou que les utilisateurs ne savent pas ce qu'ils veulent tant qu'ils n'ont pas expérimenté le système.

La difficulté d'écrire des spécifications adéquates justifie la réalisation des applications par les nouveaux outils, mettant très vite à la disposition de l'utilisateur, un prototype des systèmes demandés afin qu'il se rende compte en détail de ce qu'il veut et qu'il ait la possibilité de modifier dynamiquement le logiciel jusqu'au moment où celui-ci réponde entièrement à ses besoins.

Quoique ces méthodes fonctionnent bien pour la plupart des traitements informatiques ordinaires, elles ne conviennent pas pour des systèmes techniques très complexes. Ainsi nous croyons que les deux systèmes coexisteront:

- ceux qui exigent la modification dynamique et commandée par l'utilisateur des exigences formulées après que le système ait été initialement réalisé

- et les systèmes qui exigent avant leur réalisation une analyse et des spécifications complètes et formelles de ces exigences.

1.2. Les besoins de l'ARBED

1.2.1. La situation à l'ARBED

La dénomination ARBED S.A. désigne un groupe d'envergure internationale dont la plus grosse part de l'activité se situe dans le créneau de la sidérurgie. Même si aujourd'hui le choix n'est plus aussi catégorique, les grandes entités de ce groupe ont opté pour une informatique lourde et centralisée. Ainsi, l'on retrouve quelques centres dotés de calculateurs universels de marque IBM, homogènes pour faciliter le développement d'applications communes à plusieurs sociétés. L'environnement informatique de ces calculateurs est composé du système de gestion de bases de données IMS et du système de dictionnaire

de données, le DB/DC Data Dictionary. En effet, le service informatique a décidé de regrouper toutes les informations sur ses données sous la structure du dictionnaire de données proposé par IBM et d'en généraliser l'utilisation par tout son personnel ainsi que par tous les utilisateurs finals des applications développées.

Dans ses efforts de reconversion, le groupe contrôle un ensemble complet d'entreprises à caractère informatique (cfr. fig. 1.1.), adjoignant à ses propres services tout un know-how informatique extérieur.

La société n'est pas épargnée par la crise économique et depuis plusieurs mois une étude, appelée ALFA, est effectuée dans tous les services administratifs afin d'établir le nombre optimal de personnel et de prendre les mesures adéquates pour "Améliorer le Fonctionnement Administratif" de l'ARBED. Cette étude englobe également le service informatique et à son égard certaines mesures ont déjà été décidées pour améliorer le rendement du développement et concentrer les ressources "hardware".

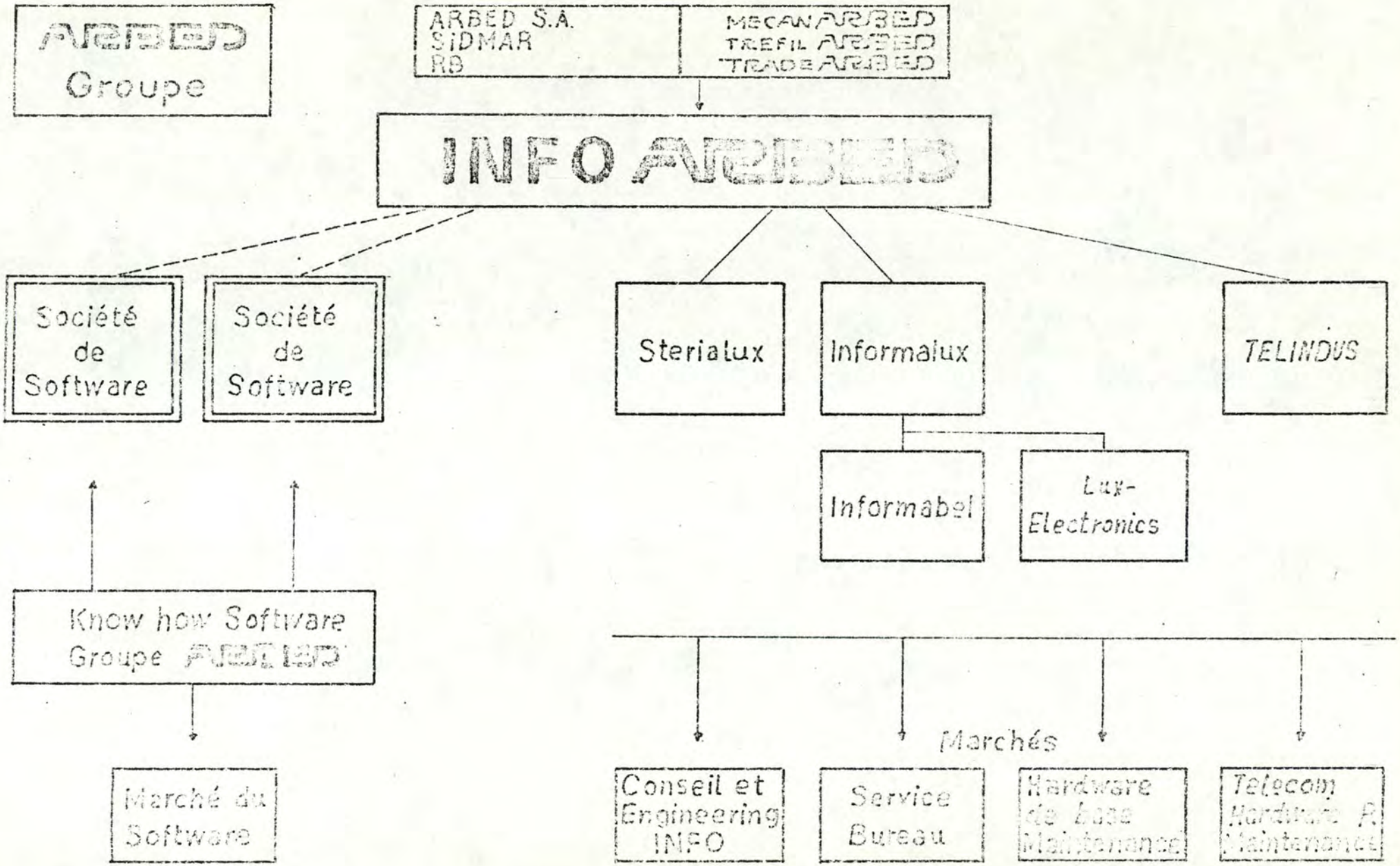


Fig. 1.1 : Organigramme des activités informatiques de l'Arbed.

Pour répondre à ces décisions, le service informatique doit fournir des efforts pour contenir le plan de charge du secteur développement dans les délais impartis. Cependant, les premières vagues de décisions de l'étude ALFA ont généré dans les autres services de nombreuses demandes à l'informatique par suite d'idées de rationalisation. Ces petits projets informatiques constituent un surplus de travail non négligeable.

L'étude ALFA a mis en évidence la complexité des relations entre le service informatique et les utilisateurs et a montré que des améliorations significatives en efficacité sont à rechercher.

L'intérêt de certaines applications (centralisation en banque de données des informations techniques, comptabilité et gestion budgétaire, un outil standard de gestion formatée de fichiers ou de tables, ...) augmenteraient de façon non négligeable si un logiciel de génération de rapports y était connecté.

De plus, des demandes de statistiques émanent aussi bien d'organismes internes (les différentes directions, ...) qu'externes (les parastataux, la Communauté Economique Européenne, ...). Jusqu'à ce jour, celles-ci ne peuvent être satisfaites qu'en les réalisant de manière manuscrite ou en désignant un programmeur pour concevoir chacune d'elle. C'est une procédure très lente dans le premier cas; lente dans le second, mais qui surcharge d'autant plus l'activité de programmation du service informatique, l'obligeant à délaisser la création de nouvelles applications proprement dites pour ces travaux routiniers et "démotivants".

Face au défi ALFA, l'informatique de l'ARBED est appelée à augmenter son rendement en développement. L'approche choisie se réalise sur deux phases bien distinctes:

- l'introduction de méthodes et de logiciels nouveaux:
  * langage de quatrième génération,
  * outil de documentation, ...

- la restructuration de la collaboration Utilisateur Service Informatique par la création d'un "service" infocentre (cfr. infra).

Pour éviter cette dernière étape de programmation dans la demande de statistiques, il a été envisagé de développer une syntaxe d'extraction et de mise en page d'informations contenues dans une base de données hiérarchisées IMS. Cet outil serait ultérieurement repris à l'Infocentre de la société.

1.2.2. Le concept d'infocentre

L'infocentre est avant tout un contexte organisationnel et une démarche méthodologique. C'est la première étape du processus inéluctable de reconquête des ressources informatiques par les utilisateurs. Il se démarque nettement de l'environnement traditionnel des services informatiques, car il entrouvre les possibilités du libre-service. Mais il reste hébergé au service informatique et contrôlé par ses professionnels dans ce sens qu'il nécessite absolument la présence de locaux et de spécialistes pour l'organisation, le conseil, l'assistance technique, la formation.

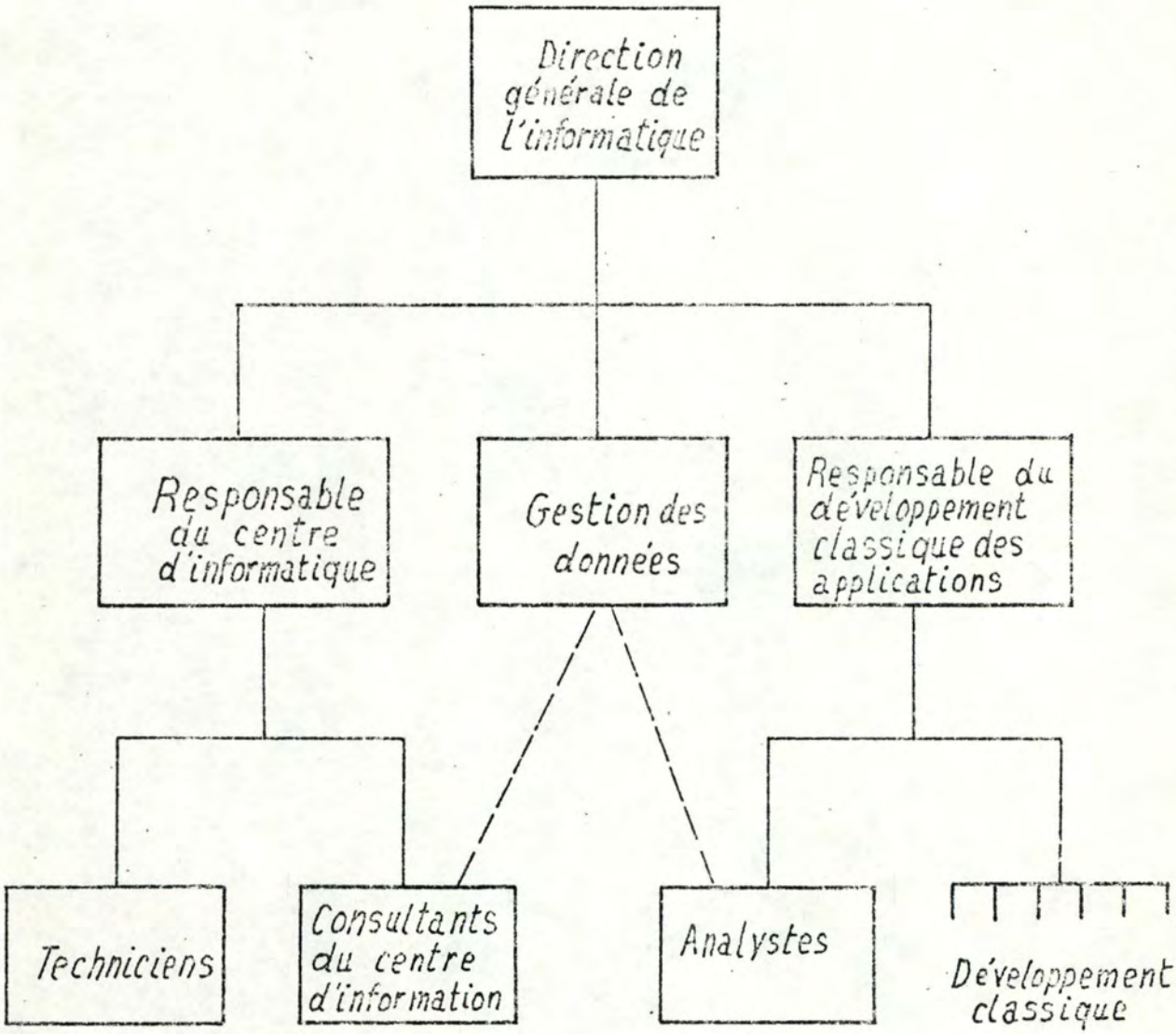
L'infocentre est un groupe conçu au sein du service informatique, pour servir directement et rapidement l'utilisateur final. Ce groupe connaît l'existence des bases de données et parfois en crée d'autres. Il diffuse cette information à l'utilisateur final en employant les types de langage décrits supra.

Ce centre doit trouver sa fonction dans l'organisation de l'informatique pour et par les utilisateurs afin:

- d'assurer que les données saisies par ceux-ci soient employées avec le maximum d'efficacité plutôt que de rester isolées,

- de les aider à développer les applications aussi efficacement que possible,

- d'encourager l'extension rapide de l'Informatique de l'utilisateur,



(traduit de [MA])

Fig. 1.2 : Organisation de l'infocentre.

- de garantir des contrôles précis et adéquats de données ,

- d'éviter la redondance dans la création des applications,

- d'éviter des problèmes d'intégrité provoqués par des mises-à-jour multiples des données,

- d'assurer que les systèmes réalisés soient contrôlables et sûrs.

L'organisation de l'infocentre peut se présenter de la façon suivante. Le traitement de l'information est subdivisé en deux: d'une part le développement conventionnel, d'autre part, l'infocentre proprement dit. Les deux sont liés à la fonction de gestion des données, qui joue un important rôle en normalisant les données qui doivent circuler entre les deux domaines. Le centre dépend du directeur de l'informatique. Son équipe se compose de consultants qui travaillent avec les utilisateurs et de spécialistes connaissant particulièrement les produits de ces derniers. Cette équipe forme et assiste les utilisateurs, crée des applications pour eux et, si possible, les encourage à résoudre leurs propres problèmes avec leur propre langage.

Chapitre 2: Le Modèle d'Accès Généralisé

2.1. Introduction

Le Modèle d'Accès Généralisé (MAG) permet de décrire les structures de données et les caractéristiques d'accès aux données, comme le permettent les Systèmes de Gestion des Bases de Données (SGBD).

Celui-ci (MAG) décrit des propriétés d'accès et admet de la redondance dans les données. On peut inclure dans un schéma d'accès des considérations d'efficacité. Aussi, il n'est pas de type conceptuel; et d'autre part, si le modèle décrit les possibilités d'accès, il ne décrit en aucun cas l'implémentation et ne se situe donc pas non plus à un niveau physique. Il faut situer ce modèle à un niveau intermédiaire appelé le niveau logique, connu par les programmes utilisateurs.

Ce modèle d'accès généralisé a été conçu par Jean-Luc HAINAUT à l'Institut d'Informatique à Namur pour être un outil puissant dans différents champs d'utilisation. Il permet d'enseigner les concepts fondamentaux des fichiers et bases de données sans se référencer à un SGBD particulier. Les concepts de chaque SGBD peuvent être décrits dans les termes du MAG, facilitant les comparaisons entre-eux. En plus, ce modèle permet de construire un schéma logique d'accès qui est indépendant d'un SGBD donné.

Dans notre étude, ce point de vue d'indépendance à un SGBD particulier (en l'occurrence IMS) explique toute l'inspiration qui nous a conduit à la définition de notre langage d'extraction.

2.2. Les objets du modèle d'accès généralisé

2.2.1. L'article.

L'article est une unité d'information mémorisée qui peut faire l'objet d'un accès, d'une création, d'une modification ou d'une suppression. Chaque article peut être identifié parmi tous les autres articles de la base de données (BD).

L'article constitue l'unité logique de transfert entre la BD et le programme d'application.

2.2.2. Le type d'article

Tout article appartient à un et un seul type qui définit les propriétés générales. Un type d'article porte un nom qui l'identifie parmi tous les types d'articles de la BD.

2.2.3. La valeur d'item

La valeur d'item est une donnée appartenant à un type de données et qui peut être manipulée par un programme d'application.

2.2.4. L'item

Les propriétés des valeurs d'item sont décrites par les items auxquels elles sont associées. L'item est un type d'information défini par un ensemble de valeurs. Un item est associé à au moins un type d'article. Il a un nom qui l'identifie parmi tous les items rattachés au même type d'article.
Deux items sont comparables s'il est possible, pour chaque couple de valeurs prises dans les ensembles de définition respectifs, de décider si ces valeurs sont égales, et éventuellement de déterminer leur ordre.

La notion d'item est munie de certaines propriétés. Un item peut être élémentaire ou décomposable. Du point de vue de sa signification, la valeur d'un item élémentaire est atomique. La valeur d'un item décomposable est constituée d'une liste de valeurs significatives, chacune d'elles appartenant à des items simples et/ou décomposables.

L'item peut être simple ou répétitif. Il est simple, si à chaque article n'est jamais associée plus d'une valeur de cet item et répétitif dans le cas contraire. La répétition peut être fixe (le même

nombre de valeurs est associé à chaque article), limitée (comprise entre 0 et un maximum déclaré) ou illimitée.

L'item peut être obligatoire ou facultatif. Il est obligatoire, si à chaque article de ce type est associée une valeur au moins. Il est facultatif s'il est permis de ne pas associer une valeur de cet item à l'article du type donné.

2.2.5. Le chemin d'accès

Le chemin d'accès fournit un mécanisme qui associe un article dit origine, à zéro, un ou plusieurs articles dits cibles, et ceci d'une manière telle qu'il soit possible, à partir de cet article origine, d'accéder successivement aux différents articles cibles associés.

Un chemin vide ne contient que l'article origine.

2.2.6. Le type de chemin d'accès

Tout chemin appartient à un et un seul type qui en définit les propriétés générales. Un type de chemin se caractérise par les types d'articles auxquels doivent appartenir les articles origines et les articles cibles. Un type de chemin peut porter un nom. A tout moment, il y a autant de chemins d'accès d'un type donné qu'il y a d'articles des types origines.

Un chemin d'accès est caractérisé aussi par les contraintes sur les modes d'insertion d'un article dans un chemin et sur les possibilités de retrait d'un article d'un chemin.

Un type de chemin peut

- être défini pour un type d'article origine et un type d'article cible, ou

- être composé de plusieurs types d'articles cibles, ou

- admettre plusieurs types d'articles origines, ou

- combiner les deux dernières possibilités: un type de chemin à plusieurs types d'origines et plusieurs types de cibles, ou

- admettre un même type d'article pour origine et pour cible.

Les types de chemin possèdent certaines propriétés. La propriété de connectivité définit le nombre maximum de cibles dans un chemin ainsi que le nombre maximum de chemins dans lesquels un article peut être cible. Ainsi les types de chemin sont classés en quatre catégories:

- les types de chemin M-N: un chemin peut contenir plus d'un article cible et un article peut être cible dans plusieurs chemins de ce type.

- les types de chemin 1-N: un chemin peut contenir plus d'un article cible et un article ne peut être cible que dans un seul chemin de ce type.

- les types de chemin N-1: un chemin ne peut contenir qu'un seul article cible et un article peut être cible dans plusieurs chemins de ce type.

- les types de chemin 1-1: un chemin ne peut contenir qu'un seul article cible et un article ne peut être cible que dans un seul chemin de ce type.

Un type de chemin peut être fort ou faible. Un type de chemin est déclaré fort pour l'un de ses types d'article cible (origine) si tout article de ce type doit à tout moment être cible (origine) d'au moins 1 chemin non vide de ce type.
L'appartenance à un chemin d'accès est automatique ou manuelle. Cette propriété spécifie que lors de la création d'un article de ce type, celui-ci sera automatiquement (automatique) inséré dans un chemin non-vide de ce type ou ne le sera pas (manuelle).

L'appartenance peut également être fixe, obligatoire ou facultative. Cette propriété spécifie que lorsqu'un tel article a été inséré dans un chemin non vide de ce type,

- soit il ne pourra plus jamais quitter ce chemin (fixe),

- soit il ne pourra plus jamais quitter un chemin de ce type tout en pouvant être transféré (obligatoire),

- soit, à tout moment, il pourra être détaché de ce chemin et éventuellement être réinséré dans un autre chemin du même type (facultatif).

On peut définir un chemin inverse à un autre. Si le chemin ch1 est inverse à ch2, alors:

- pour tout vj cible d'un chemin ch1 d'origine Pi, il existe un chemin ch2 d'origine vj et de cible Pi.

- ch2 est inverse à ch1.

Il peut arriver que l'intersection de deux chemins ne contienne pas plus d'un article. On peut dire alors que le groupe de chemins est identifiant pour le type d'article.

2.2.7. L'article système

Toute base de données contient un et un seul article particulier, appelé article système. Il constitue un point d'entrée privilégié dans la base de données. Il peut être origine de chemins d'accès, mais non cible. Celui-ci représente l'existence de la base de données et porte en général le nom de la base de données.

2.2.8. Le fichier

Un fichier est une collection dynamique d'articles. A un fichier sont associés un ou plusieurs types d'articles. Un type d'article est associé à au moins un fichier; un article appartient à un et un seul fichier.

2.2.9. La base de données

Une base de données est la collection de tous les articles qui, à un instant déterminé, décrivent complètement un système réel. Une base de données est constituée d'un ou plusieurs fichiers. Un fichier ne peut appartenir à plus d'une base de données.

2.2.10. La clé d'accès

Une clé d'accès est un item (ou une liste d'items) d'un type d'article tel qu'il existe un mécanisme qui permette d'accéder successivement aux articles auxquels est associée une valeur déterminée de cette clé.

Une clé d'accès est caractérisée par le référentiel (un sous-ensemble d'articles de la base de données) dans lequel elle est définie.

On retient les référentiels suivants:

- tous les articles de la base de données,
- tous les articles d'un type dans la base de données,
- tous les articles d'un fichier,
- tous les articles d'un type dans un fichier,
- tous les articles d'un chemin d'accès, et
- tous les articles d'un type dans un chemin d'accès.

2.2.11. Item identifiant

Nous allons revenir à la notion d'item pour définir le concept d'identifiant. Un identifiant d'un type d'article est un item ou un groupe d'items tel qu'il n'existe pas dans le référentiel donné, plus d'un article qui soit associé à une même valeur de cet item ou du groupe d'items.

2.2.12. L'ordre

La plupart des primitives d'accès aux données offrent un accès séquentiel aux articles d'un référentiel donné dans la base de données, selon un certain ordre.

2.2.13. La référence

Une valeur spéciale, contrôlée de manière interne, est associée à chaque article de la base de données. Bien que n'étant pas véritablement un item, ces valeurs peuvent être considérées comme constituant un identifiant dans le référentiel: tous les articles de la base de données.

2.3. Les primitives du modèle d'accès généralisé

La description d'un type de données n'est complète que lorsque l'on a énuméré et spécifié les opérations permises sur les données de ce type. Les primitives sont les opérations qu'un programme peut réaliser sur les données décrites dans la partie précédente.

On distingue trois classes de primitives:

- les primitives d'accès, qui mettent en oeuvre les mécanismes d'accès.
- les primitives de mise-à-jour, qui font évoluer l'état de la BD,
- les primitives de contrôle d'environnement qui, fonctionnellement, permettent la création de super-primitives, l'établissement de points de reprise et la maîtrise de la concurrence.

2.3.1. Les primitives d'accès

Les différents modes d'accès aux articles d'une base de données peuvent être décrits comme des variantes d'un principe unique: l'accès aux articles d'une séquence. Toute primitive d'accès à des articles peut en effet être définie pour:

- la séquence d'articles dans laquelle elle agit,
- la position dans la séquence de l'article demandé.

On distingue quatre classes principales de primitives selon le référentiel auquel elles se rapportent:

- l'accès séquentiel aux articles de la base de données,
- l'accès séquentiel aux articles d'un fichier
- l'accès séquentiel aux cibles d'un chemin d'accès
- l'accès séquentiel aux articles qui ont une certaine valeur pour une clé d'accès donnée.

Il faut envisager des ordres différents selon qu'il y a un seul type d'article ou plusieurs types d'articles dans le référentiel.

Si le référentiel contient un seul type d'article, on retiendra les quatre types d'ordre suivants:

- ordre non-significatif
- ordre lié au moment d'insertion: chronologique ou antichronologique
- ordre trié selon les valeurs d'une clé de tri: croissante ou décroissante
- ordre programmé: la position d'un article est choisie par le programme après ou avant un certain article.

Si le référentiel contient au moins deux types d'articles différents, on retiendra les types d'ordre suivants:

- tout le référentiel est ordonné selon un des quatre types définis ci-dessus, sans tenir compte des types d'articles;

- un tri majeur est maintenu sur le type d'article, de façon à ce que tous les articles d'un type soient regroupés; à l'intérieur de chaque type d'article est appliqué un des quatre types d'ordre définis ci-dessus;

- aucun ordre n'est maintenu entre des articles de différents types, mais pour les articles d'un même type, on applique un des quatre types d'ordre définis ci-dessus.

2.3.1.1. Accès à la base de données

Les deux primitives délimitent dans le temps le travail sur une base de données.

Ouverture d'une base de données: elle permet de rattacher un utilisateur (programme) à la base de données s'il y est autorisé; l'utilisateur doit préciser le niveau de protection qu'il demande.

Fermeture de la base de données: elle libère la base de données ainsi que les restrictions éventuelles imposées lors de l'ouverture.

2.3.1.2. Accès aux fichiers

Ouverture d'un fichier: ceci correspond à l'ouverture classique des fichiers; ici aussi, l'utilisateur doit préciser le niveau de protection qu'il demande. Avant d'accéder à un article, le fichier correspondant doit être ouvert.

Fermeture d'un fichier: cette primitive libère le fichier ainsi que les restrictions éventuelles imposées lors de l'ouverture.

2.3.1.3. Accès aux articles cibles d'un chemin

Trois types d'accès primitifs sont généralement admis:

- le parcours séquentiel direct, par lequel les articles cibles sont obtenus du premier au dernier selon l'ordre défini pour le type de chemin;

- le parcours séquentiel inverse, par lequel les articles cibles sont obtenus du dernier au premier selon l'ordre inverse de celui qui est défini pour le type de chemin;
- l'accès direct à l'article cible de rang déterminé.

Les articles concernés sont tous les articles du chemin , ou exclusivement ceux d'un type déterminé; dans ce dernier cas, il convient de préciser ce type d'article comme argument des primitives.

Les primitives proposées permettent d'accéder:

- au premier, au dernier article cible d'un chemin;
- à l'article cible suivant, ou précédent dans un chemin;
- directement à un article dans un chemin.

2.3.1.4. Accès à des articles par clé d'accès

La clé d'accès peut être considérée comme un cas particulier du chemin d'accès, mais on lui associe de préférence des primitives spécifiques.

D'une manière générale, les référentiels d'une clé d'accès sont un chemin d'accès, celui-ci représentant à l'occasion une séquence des articles d'un fichier ou de la base de données.

Les principes proposés sont:

- l'accès au premier ou dernier article associé à une valeur de clé d'accès,
- l'accès à l'article suivant ou précédent associé à une valeur de clé d'accès,
- l'accès direct lié à une clé d'accès.

2.3.1.5. Accès lié aux identifiants internes

De nombreux systèmes de gestion associent à chaque article un identifiant interne qui est aussi une clé d'accès. Ils proposent généralement deux fonctions: l'accès à l'article d'identifiant donné et l'accès à l'identifiant d'un article.

2.3.1.6. Accès aux valeurs d'item d'un article

L'article ayant une existence autonome par rapport à ses valeurs d'item, il est nécessaire de prévoir une primitive d'accès à ces valeurs.

2.3.2. Les primitives de modification

Une opération de modification a pour effet de faire passer l'ensemble des données d'un état dans un autre de manière que cet ensemble continue à refléter à tout instant l'état du monde réel à représenter. Elle doit garantir la cohérence de l'ensemble des données, c'est-à-dire que l'opération doit respecter complètement la structure et les contraintes diverses définies sur ces données.

Aussi, voici des primitives nécessaires:

2.3.2.1. Création d'un article

Si les contraintes d'intégrité définies dans le schéma peuvent être vérifiées à l'aide des données en entrée, cette fonction enregistre dans la base de données un article en accord avec le mode de stockage du type d'article spécifié et l'insère dans les chemins d'accès automatiques.

2.3.2.2. Suppression d'un article

Etant donné la référence d'un article, cette fonction retire cet article de tous les chemins d'accès dont il est cible, détruit les chemins d'accès dont il est origine, supprime éventuellement les cibles de ces chemins et enfin retire l'article du fichier.

2.3.2.3. Modification des valeurs d'item d'un article

Etant donné la référence d'un article, le nom des item et les valeurs d'item, cette fonction ajoute de nouvelles valeurs à l'article, retire des valeurs ou change des valeurs de l'article.

2.3.2.4. Insertion d'un article dans un chemin

Etant données la référence de l'article et la référence du chemin d'accès, cette fonction insère l'article dans le chemin d'accès. Un point d'insertion peut être spécifié.

2.3.2.5. Retrait d'un article d'un chemin d'accès

Etant données la référence de l'article et celle du chemin d'accès, la fonction retire l'article du chemin d'accès.

2.3.2.6. Changement de chemin d'accès

Etant données la référence d'un article et les références de deux chemins d'accès du même type, cette fonction retire l'article du premier chemin d'accès et l'insère dans le deuxième.

2.3.3. Les primitives de contrôle

Un fonctionnement correct entre une base de données et plusieurs utilisateurs doit être garanti par des solutions aux problèmes de concurrence, de reprise et de consistance des données. Ces problèmes se situent par rapport aux contraintes d'intégrité qui ne sont pas directement prises en charge par les SGBD's. Des réponses sont apportées par certaines primitives. Ainsi, le début d'une séquence atomique:

marque un moment à partir duquel les modifications apportées à la base de données sont réversibles jusqu'à ce que la primitive de fin de séquence atomique soit réalisée.

- la fin d'une séquence atomique:

  les modifications apportées à la base de données depuis le dernier "début d'une séquence atomique" sont considérées comme définitives.

- l'échec d'une séquence atomique:

  les modifications apportées à la base de données depuis le dernier "début d'une séquence atomique" sont retirées de la base de données.

A l'aide de ces trois primitives, le programmeur peut concevoir des primitives de haut niveau pour faire face aux problèmes soulevés supra.

CHAPITRE 3: La notion de dictionnaire de données

3.1. Les modèles sémantiques

Le développement considérable de l'informatique en général a permis d'étudier des problèmes de plus en plus complexes. Cette complexité croissante a entraîné un besoin de documentation sur toutes les étapes du cycle de vie d'un projet informatique. Ce besoin est exprimé par des utilisateurs de formation différente, aux responsabilités distinctes et qui désirent les informations en termes différents.

Bien que l'information ait toujours connu ses lettres de noblesse dans l'entreprise, celle-ci était mal gérée. Chaque service possédait sa propre collection d'informations et ignorait le travail des autres. La redondance des données se trouvait non-contrôlée, tout en étant en contradiction avec la volonté d'informatiser des applications toujours plus complexes.

Pour répondre à ce manque de gestion des informations, il fallait introduire un outil dans les systèmes, mais cela devait se produire de manière réfléchie. En effet, durant l'évolution des méthodes hiérarchiques, de réseaux et relationnelles dans la construction des BD's, il est devenu graduellement apparent que la construction d'une BD devait être développée de manière indépendante des applications. Ainsi apparaît la notion de schéma conceptuel pour modéliser la BD indépendamment des applications, ainsi que les schémas externes dérivés du schéma conceptuel pour exprimer des exigences de données spécifiques à une application.

Cette volonté de modéliser indépendamment des applications a produit un éventail de modèles sémantiques de données, donc des dictionnaires de données. Un de ceux qui a connu le plus de succès est le modèle "entité - association" qui fournit des constructions de données à deux niveaux: le niveau conceptuel, dont les constructions incluent des entités, des relations, des ensembles de valeurs et des attributs; et le niveau de la représentation où les constructions conceptuelles sont transformées en tables.

Nous allons présenter dans ce chapitre les notions du concept de dictionnaire de données, qui de manière synthétique peut être défini comme un répertoire de données concernant des données (DATA about DATA, META-DATA).

3.2. La définition

Un dictionnaire de données se définit comme étant un système informatique qui permet, ou du moins a pour but de décrire un système d'information, et éventuellement d'aider à la conception et à la maintenance de celui-ci.

Comme tout système informatique, le dictionnaire de données est composé des trois composants suivants:

1) Les données qui sont des descriptions:

   * des données reliées à la base de données, aux fichiers conventionnels, à des informations non-automatisées.

   * des processus automatisés ou non-automatisés.

   * des utilisateurs en tant que personne ou organisation

2) des processus qui permettent:

   * de remplir le dictionnaire de données et de le maintenir dans un état cohérent

   * la production de rapports

   * de générer, éventuellement, des bases de données et fichiers à partir des descriptions.

3) des utilisateurs qui le consultent et mettent à jour des applications à chaque étape du cycle de vie, quelle que soit leur formation.

3.3. L'évolution des dictionnaires de données

3.3.1. Un complément aux systèmes de gestion des bases de données

L'introduction des bases de données et des techniques associées a mis en évidence tous les bénéfices qu'apporte une seule source de données vis-à-vis de fichiers disparates. Mais les systèmes de gestion des bases de données (SGBD) n'apportent pas aux utilisateurs toutes les facilités de gestion qu'ils pourraient en attendre. Aussi, les dictionnaires de données ont été développés après l'apparition des SGBD's afin de subvenir aux besoins non pris en charge par ceux-ci.

Nous n'en citerons que les principaux:

- la redondance des données,
- le manque de standardisation,
- le manque de connaissance des informations dérivées de données.
- le manque d'outils permettant d'estimer le coût de la modification d'une base de données.

En effet, tout utilisateur a besoin d'informations concernant son activité, à n'importe quel niveau que cela soit. Le traitement de l'information nécessite des données sur les données. C'est ce que le dictionnaire de données fournit: il remplace la documentation disparate par un stockage d'informations sur les données utilisées: items, articles, fichiers, applications, programmes ainsi que leurs "interrelations". Il pourra fournir également des renseignements sur la méthode de stockage des données, leur signification, leur organisation, les méthodes d'accès et de mise à jour.

Par l'amélioration de la documentation qu'il apporte, le dictionnaire de données favorise le contrôle. Les données y sont référencées de manière standard. Ce contrôle ne se limite pas à la redondance, car le dictionnaire de données fait apparaître les circonstances dans lesquelles tel article peut être accessible et mis à

jour. Ainsi, il autorise un contrôle de l'utilisation des données. Cependant, son utilisation effective dépendra de sa complétude à tout instant, car il doit être la source de TOUTES les informations concernant les données traitées.

Pour assurer cette complétude, les méta-données peuvent être insérées dans le DD manuellement, mais une meilleure solution est un "pont" automatique des autres sources de méta-données vers le DD et mieux encore est un pontage dans l'autre sens. L'existence de ces primitives va caractériser les différentes générations de DD par le rôle passif, actif ou dynamique qu'il supporte.

3.3.2. Le rôle passif d'un dictionnaire de données

Le rôle passif d'un DD est de prendre en charge la documentation de manière automatique. Les références aux données par l'intermédiaire du DD permettent l'élaboration du schéma conceptuel à partir duquel l'étape d'implémentation est développée. Le DD passif donne les moyens de rassembler l'information essentielle au développement des systèmes, même complexes. Il permet de résoudre les conflits de données par la construction de définitions claires, uniques et non-ambïguës.

3.3.3. Le rôle actif d'un dictionnaire de données

Le DD passif n'est qu'un manuel de références implémenté sur ordinateur, mais le DD n'assure pleinement ses fonctions que s'il est utilisé. Pour assurer la bonne utilisation d'un DD, il doit devenir une partie active du système d'informations en fournissant une aide directe au design de la structure de données, des BD et des systèmes: cette aide constitue le rôle actif d'un DD. La structuration des données s'y fait par une famille de modèles. Pour arriver à l'analyse de chaque structure, des outils permettent de développer des structures sous-parentes à une vue utilisateur ou de synthétiser un modèle de cette structure de données. Des possibilités d'évaluation du contenu sont fournies pour analyser les méta-données (homonymes, synonymes).

3.3.4. Le rôle dynamique d'un dictionnaire de données

Celui-ci consiste à améliorer l'aspect actif du DD. Le DD actif interagit avec les programmes au moment de la compilation, tandis que le DD dynamique interagit avec les programmes au moment de l'exécution.

Le rôle dynamique interagit directement avec les softwares pour indiquer où l'information peut être obtenue. L'utilisateur précise dans un langage de requête sa demande, qui sera transformée automatiquement au niveau de l'implémentation. Ainsi, le DD détermine où se trouve l'information et la fournit.

3.4. Les fonctions d'un dictionnaire de données

Certaines fonctions du DD doivent être mises en évidence:

1. L'enregistrement d'informations tant au niveau conceptuel, d'implémentation, que du niveau opérationnel.

2. La vérification de l'uniformité et la validation des définitions; le DD ne doit pas seulement enregistrer des faits, mais vérifier qu'ils sont validés.

3. La gestion des versions multiples. Lorsqu'un changement se produit, le DD doit pouvoir être mis à jour pour produire une nouvelle version du schéma. Il doit être possible de garder et de différencier les différentes versions d'une vue:
   - comme référence historique
   - parce que celle-ci peut ne pas encore être opérationnelle,
   - parce que celle-ci peut n'être utilisée que dans un but de tests.

4. Les transformations. Etant en possession d'un schéma conceptuel, le système devrait être capable de faciliter le passage à la vue au niveau

de l'implémentation. Possédant cette vue contenant le schéma, les sous-schémas, les contraintes d'utilisation, il devrait être possible de passer à une vue opérationnelle.

5. Des rapports et interrogations on-line.

6. La génération de description de programmes et de données à partir d'informations concernant le niveau d'implémentation.

7. La sécurité. Le DD est un répertoire idéal pour les informations en ce qui concerne l'aspect confidentiel et le contrôle d'accès aux données.

8. L'analyse de l'impact des modifications. La connaissance du contenu des fichiers en items et leur utilisation par tel programme permet de déduire quels programmes seront affectés par une modification d'item.

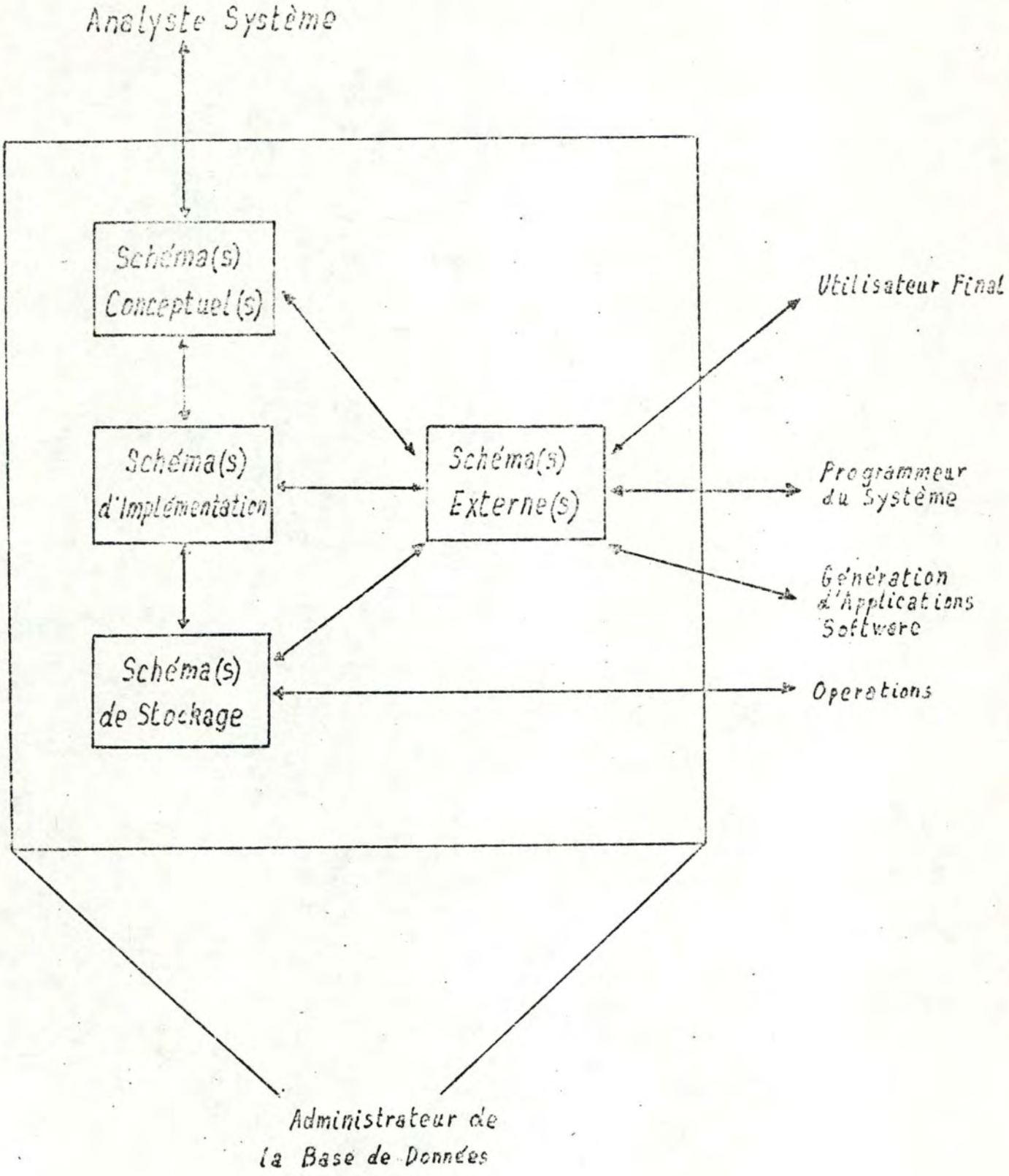
9. L'ordonnancement et le contrôle de jobs. Certaines facilités permettent d'alléger les problèmes associés aux opérations manuelles: contrôle de la séquence des programmes dépendants et mesures de sécurité nécessaires à maintenir l'intégrité des données.

Cette liste n'est pas exhaustive, mais permet de se rendre compte de l'apport du dictionnaire de données dans un système d'informations complexe.

3.5. L'architecture d'un dictionnaire de données

3.5.1. Les caractéristiques générales

Les caractéristiques des descriptions de données à maintenir dans un DD peuvent varier à la fois dans leurs usages et leurs parties. Le système du dictionnaire de données doit posséder les moyens de maintenir les différentes vues relatives au système d'information. Les niveaux de description sont les suivants:



(traduit de [BSC])

Fig. 3.1: Les relations entre les différentes vues du D.D.

- le schéma conceptuel est une description des données et des exigences du système de traitement indépendamment de considérations d'implémentation.

- le schéma d'implémentation est une description de la base de données au niveau des enregistrements logiques indépendamment des langages hôtes de programmation qui peuvent être utilisés pour traiter la BD.

- le schéma de stockage est une description qui permet de déterminer la structure physique de la BD et la localisation physique de la donnée.

- le schéma externe est une description d'un sous-ensemble de la BD, des chemins d'accès recommandés, des processus autorisés ou non à utiliser le schéma externe.

La figure fig. 3.1. représente les différentes relations entre ces vues au sein du DD et les rapports existant entre les utilisateurs et ces différentes vues.

3.5.2. Les communications entre les différentes vues du DD

Les communications entre les différents types d'utilisateurs peuvent être produites par des interfaces entre les différents niveaux du DD. Nous allons présenter chacun d'eux en les justifiant.

3.5.2.1. L'interface entre le schéma conceptuel et le shéma externe

Celle-ci génère des vues utilisateurs de la base de données. En effet, la vue utilisateur peut être introduite ou générée à partir du schéma conceptuel. Une part essentielle du schéma conceptuel est de décrire les utilisateurs d'une BD et les noms avec lesquels un utilisateur réfère les éléments de la BD.

Du fait que l'utilisateur peut introduire lui-même sa vue, cette interface devra être suffisamment puissante de manière à simplifier au maximum la tâche de l'utilisateur.

3.5.2.2. L'interface entre le schéma conceptuel et le schéma d'implémentation

Celle-ci peut être utilisée pour dresser les exigences de la BD dans le schéma d'implémentation. La première fonction du schéma conceptuel est d'agir comme une aide au design du système en utilisant la BD décrite par le schéma d'im- plémentation.

L'interface peut être utilisée comme moniteur au design de la BD.

3.5.2.3. L'interface entre le schéma d'implémentation et celui de stockage

Quand un changement de design de l'implémentation est proposé, il est possible d'utiliser une transformation entre les schémas d'implémentation et de stockage pour évaluer l'impact d'un changement de design.

3.3.2.4. L'interface entre le schéma d'implémentation et le schéma externe

Celle-ci fournit au programmeur des procédures d'accès aux données de manière à ce que celui-ci ne soit plus obligé de programmer sa propre séquence de primitives d'accès. L'interface sera également un outil à la génération de schémas externes.

3.5.2.5. L'interface entre le schéma externe et le schéma de stockage

Celle-ci constitue un lien entre les programmes et les structures physiques de données.

CHAPITRE 4: Les spécifications

4.1. Introduction

Comme nous l'avons déjà dit dans le premier chapitre, l'organisation en infocentre s'explique à l'ARBED par les besoins des utilisateurs, et le nombre, de plus en plus grandissant de demandes d'informations. En effet, jusqu'à présent, le temps de réponse du service informatique à de telles demandes est de l'ordre d'un à deux mois. L'idée est donc de libérer le service informatique de toute création de statistiques nécessaires tant à l'intérieur qu'à l'extérieur de la société.

Vu le délai accepté jusqu'à présent, il est admis que ces rapports puissent être réalisés en grande partie en dehors des heures de bureau.

Ce que nous voulons essentiellement présenter dans ce chapitre peut se résumer en une présentation claire des objectifs poursuivis, une définition précise d'un rapport tel qu'il est vu dans l'entreprise et une présentation des formes sous lesquelles la requête peut être formulée.

4.2. Les objectifs poursuivis

Les objectifs poursuivis peuvent se résumer en trois points:

- l'extraction de données disponibles dans n'importe quelle BD IMS de la société suivant des critères de sélection que l'utilisateur peut aisément définir,

- la transformation de certaines ou de toutes les données extraites par des formules mathématiques plus ou moins complexes déterminant des résultats,

- l'impression de ces résultats sous une forme librement choisie par l'utilisateur.

Pour arriver à ces objectifs, il faut gérer toute la vie d'un rapport. De plus, le système ne peut autoriser n'importe quel utilisateur à accéder à n'importe quelle information de l'entreprise. La sécurité et le caractère confidentiel des données doivent être maintenus. Aussi certaines fonctions secondaires au but du système doivent exister.

L'utilisateur doit pouvoir être identifié et autorisé à pénétrer dans le système. Aussi, cette autorisation de chacun des utilisateurs doit pouvoir être gérée, c'est-à-dire qu'il doit exister des modules de création, de suppression, de visualisation, de modification d'informations concernant l'utilisateur et d'affichage de la liste des définitions de rapport auxquelles chaque utilisateur est autorisé à accéder.

La définition de rapport doit être composée. Cette fonction consiste en l'édition et l'analyse de la définition de rapport. Cette composition est également gérée (création, suppression, modification) afin de permettre une mise-à-jour de la BD des définitions de rapports.

Un même rapport peut être demandé pour quelques valeurs connues de quelques variables. Ces variables prennent le nom de paramètre. Cette notion doit être également gérée afin de pouvoir en répéter l'usage tout en réduisant le travail d'introduction de ces paramètres.

L'utilisateur minutieux peut désirer se rendre compte visuellement de la structure d'un rapport avant d'en poursuivre l'élaboration. Aussi, il peut faire apparaître la "charpente", le "squelette" d'une définition de rapport. Ce dernier consiste en un output à l'écran ou sur liste de la définition de rapport sans y avoir rempli les zones réservées aux informations des bases de données, ainsi que celles réservées aux paramètres. Autrement dit, c'est une représentation des caractéristiques communes à toute exécution d'une définition de rapport.

L'exécution d'un rapport peut se dérouler de telle manière que le format horizontal de l'output généré s'avère supérieur à celui de l'écran et / ou de la liste. Dès lors, le rapport doit être considéré comme la concaténation, tant en hauteur qu'en longueur, de sous-rapports pouvant contenir sur un écran ou sur une page de liste. Pour la sortie de son résultat, l'utilisateur doit définir un ordre d'apparition en séquence des

sous-rapports: soit de haut en bas puis de gauche à droite, soit de gauche à droite puis de haut en bas.

Quand l'utilisateur désire exécuter une définition de rapport, il pourra choisir entre:

- une exécution batch pour une définition de rapport et un fichier de paramètres spécifiques ou par défaut de tout fichier de paramètres lié au rapport.

- une création d'un fichier de paramètres qui peut être suivie d'une exécution on-line ou batch.

4.3. La définition de rapport à l'ARBED

Voici la manière dont est vu un rapport.

Une statistique est un tableau bidimensionnel composé d'un nombre, à priori inconnu, de lignes imprimées et / ou affichées. Une ligne appartient à un certain type dont les caractéristiques se traduisent par une découpe très précise en colonnes logiques, du nombre total de positions physiques disponibles. Le nombre et la définition des types est propre à une statistique.

La nature de l'information que l'on retrouve dans une cellule ou élément du tableau peut être de plusieurs types définis ci-après. La constante peut être numérique ou alphanumérique. La constante est donnée au moment de la définition d'un rapport à titre définitif. Le paramètre externe est fourni avant chaque exécution du programme générant la statistique. La variable est définie en nom par programmation, une fois pour toutes, mais sa valeur instantanée est calculée par programme et est donc susceptible de changer à tout moment pendant l'exécution. La donnée est une information obtenue dans une Base de Données. Le résultat est le produit d'opération(s) mathématique(s) effectuée(s) sur des données.

Un exemple typique est représenté par la figure 4.1.

Chaque rapport peut contenir un en-tête et/ou un bas-de-page afin d'y répéter certaines informations spécifiques sur chaque page de liste ou chaque écran comme il plaira à l'utilisateur.

4.4. Quel environnement

Ayant défini le produit qui devra résulter de ce logiciel, il nous semble opportun de présenter ceux qui utiliseront cet outil et comment ils s'en serviront.

Dans l'optique du projet, l'utilisateur final ne se présentera pas nécessairement comme informaticien, mais bien comme toute personne qui désire certaines statistiques sur des données de la société auxquelles elle peut accéder. Ce besoin pourra dans certains cas être sporadique, sans pour cela que le système ne conserve pas son côté attrayant aux yeux de l'utilisateur en lui imposant un travail supérieur à celui qu'il aurait en utilisant une autre méthode.

Un utilisateur privilégié de la version finale restera le cadre d'entreprise qui saura apprécier comme moyen d'introduction dans le système en plus d'un langage formel, un langage graphique et un langage interactif par questions-réponses.

CHAPITRE 5: L'ARCHITECTURE

5.0. Introduction

Ce chapitre représente le noeud de ce mémoire. Nous le subdivisons en trois grandes parties:

- un rappel des objectifs,
- l'architecture globale: où nous présentons une architecture en suivant le cycle de vie d'un rapport,
- l'architecture du système d'exécution: où nous présentons comment l'exécution proprement dite d'un rapport se déroule.

5.1. Rappel des objectifs et contraintes

Nous sommes, du point de vue de l'architecture, quelque peu tiraillés entre deux optiques: d'un côté, nous aspirons à une architecture ouverte telle qu'elle a été proposée dans des travaux précédents effectués à l'Institut; et de l'autre côté, l'ARBED tient à posséder un produit qui 'tourne' sans trop de superflu.

Dans ce paragraphe, nous ne faisons que citer les objectifs et contraintes générales:

- le générateur de rapport travaillant en extraction de bases de données: le point le plus important est l'accès aux bases de données et non l'aspect de génération d'un rapport. On considère la génération d'un rapport comme un type particulier d'accès à des bases de données.
- l'utilisation à la fois par des programmeurs et des usagers occasionnels:

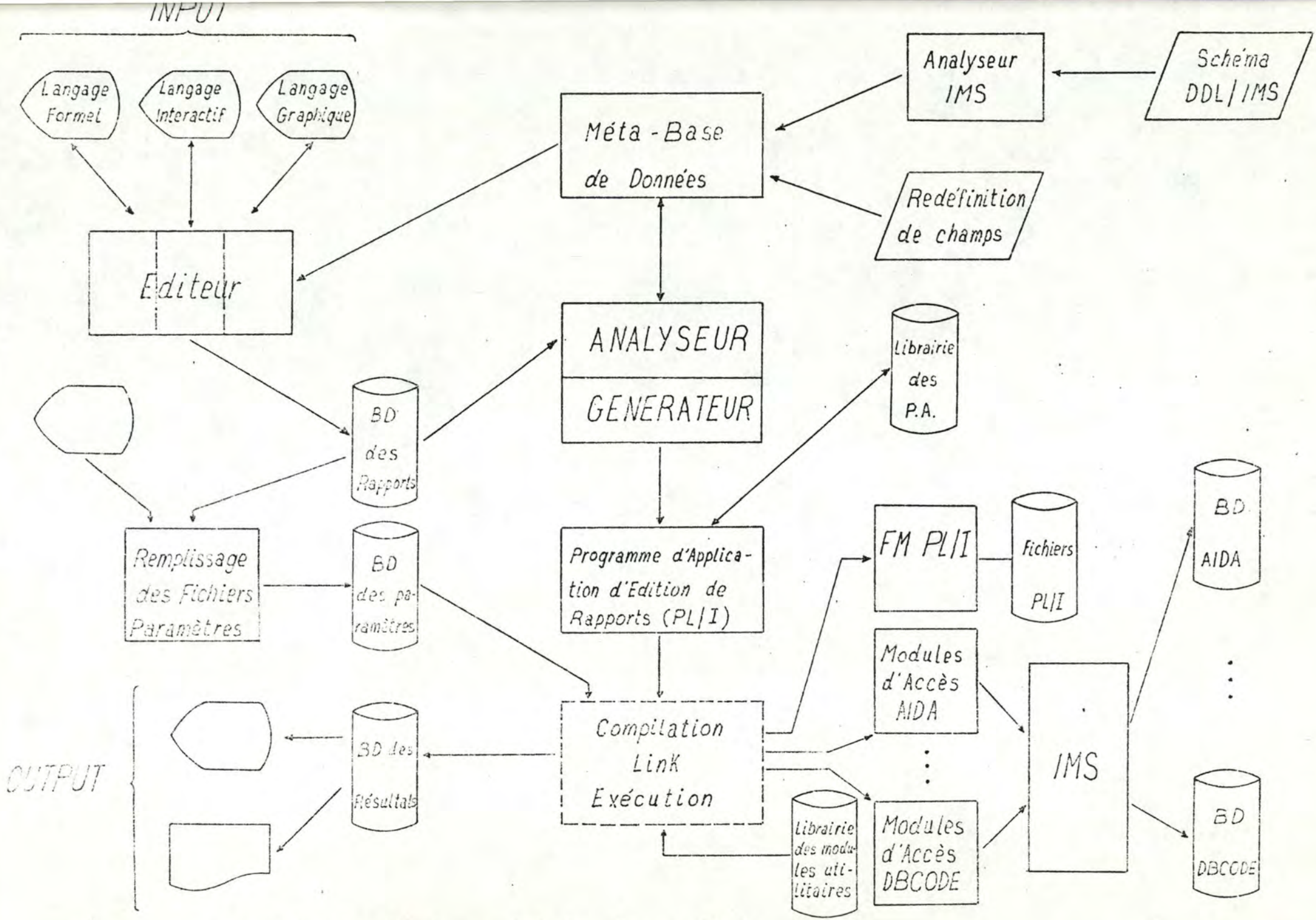


Fig. 5.1. : Architecture globale du système

il s'agit de trouver un langage de manipulation de données adapté au référentiel de l'utilisateur. Aussi, afin de permettre différents niveaux de perception d'une base de données, nous devons prévoir différents langages.

- la prise en charge de bases de données existantes.
  Cela suppose que le système soit suffisamment souple que pour s'adapter à l'environnement de chaque SGBD.

- la coexistence de bases de données hétérogènes.
  Si nous voulons éviter d'avoir à repartir de zéro pour chaque SGBD, nous devons avoir un tronc commun sur lequel se greffent des interfaces propres à chaque SGBD.

- le générateur transportable d'une machine à l'autre.
  L'usage d'un langage de haut niveau pour la programmation ainsi que d'une modularisation destinée à cacher les caractéristiques propres à une machine doivent permettre au système d'être transportable.

- le générateur extensible.

5.2. L'architecture globale

La découpe fonctionnelle du système nous donne assez facilement une architecture globale du système de génération de rapport (Fig. 5.1.). Celle-ci suit le cycle de vie d'un rapport, à savoir définition, paramétrage, exécution et gestion des résultats d'un côté, et gestion d'une méta-base de données de l'autre.

5.2.1. La définition d'un rapport

L'utilisateur crée la définition de son rapport au terminal au moyen d'un éditeur adapté, soit à un langage formel, soit à un langage interactif, soit à un langage graphique. Cet éditeur apporte le qualificatif 'user-friendly' au système en mettant à la disposition de l'utilisateur des fonctions d'aide en connexion avec la méta-base de données.

Dès que la définition de rapport est introduite, celle-ci est mémorisée dans une base de données qui permet de la gérer conjointement à la gestion des accès des utilisateurs.

5.2.2. La définition des valeurs de paramètres

La définition de rapport doit être paramétrisable. Aussi, le système doit comporter un outil qui permet de définir des fichiers de valeurs de paramètres et de les gérer d'une façon générale. Liés à une définition de rapport, ces fichiers sont conservés dans une base de données.

La fonction essentielle de cet outil est la possibilité de créer agréablement un de ces fichiers. De nombreuses manières plus 'User Friendly' l'une que l'autre peuvent être imaginées pour cette création. Cependant dans un premier temps, nous ne cherchons pas à rencontrer ce caractère amical dans cette fonction.

L'introduction se fait en répétant dans un fichier autant de fois qu'il y a de paramètres un même ordre dont la syntaxe est:

<ident-paramètre> = valeur;

où <ident-paramètre> est l'identificateur d'un paramètre.

Cette fonction peut être présentée d'une autre manière toute aussi simple. En utilisant cette fonction, l'utilisateur voit apparaître sur son écran des ordres amicaux lui proposant d'introduire des valeurs aux variables mentionnées. En effet, l'outil est capable d'aller consulter la définition de rapport pour lire la liste des identificateurs des paramètres et de les faire apparaître l'un après l'autre à l'écran.

La première solution représente, à notre avis, une réponse minimale au besoin.

Cette définition de paramètres est gérée par les fonctions habituelles de gestion de fichier (modification, destruction, visualisation, création, ...).

5.2.3. L'exécution

Après avoir utilisé la définition d'un rapport et la définition d'un jeu de paramètres, il faut provoquer l'analyse de cette première phase. Cette phase est explicitée dans les paragraphes suivants. Celle-ci s'étant terminée correctement, l'utilisateur peut lancer l'exécution de la définition de rapport (son code objet) sur les valeurs de paramètre introduites précédemment pour générer effectivement le rapport désiré. Comme pour la phase d'analyse, la phase d'exécution est explicitée infra.

5.2.4. La gestion des résultats

L'exécution étant terminée, les résultats produits sont stockés en tant qu'entité dans une base de données. Aussi, celle-ci doit être gérée en conséquence. L'utilisateur autorisé doit avoir la possibilité d'éditer sur écran ou sur liste un de ces résultats, ou détruire l'un d'eux. Si ces résultats comprennent des graphiques, un outil doit transformer leur format interne de stockage en une forme réellement graphique à l'écran ou sur liste.

Nous pouvons présenter la base de données qui supporte le système (fig. 5.6.). Le segment racine comprend les informations concernant l'utilisateur. Parmi les segments dépendants, nous trouvons un segment 'Environnement Utilisateur'. Dans celui-ci, l'utilisateur peut se définir tout un environnement qui lui est propre, qui individualise sa programmation, telle la définition d'abréviations, ... Les segments 'Définition de rapport' et 'Paramètre' représentent les notions évoquées aux paragraphes précédents.

5.3. L'architecture de l'exécution

La partie essentielle du système de génération de rapports est l'ensemble des processus permettant l'exécution proprement dite d'un rapport.

Nous allons passer en revue les décisions ayant conduit à l'architecture finale.

5.3.1. Les composants de base

L'architecture de base est proposée à la figure 5.2. Ses composants sont:

- l'expression de la requête dans un langage donné, LDA;
- une méta-base de données accessible par l'intermédiaire d'un SGBD;
- une ou des bases de données accessibles par un ou des SGBD's;
- un ensemble de programmes permettant l'exécution de la requête et la production d'un rapport:

5.3.2. Le mode d'exécution: l'interprétation ou la compilation

Le problème consiste à analyser une requête exprimée dans un langage de haut niveau afin d'en vérifier sa syntaxe et sa sémantique et puis de l'exécuter. Traditionnellement, il y a trois voies pour arriver à l'exécution:

1. la compilation de la requête dans un langage cible exécutable par une machine disponible, puis l'exécution proprement dite;

2. l'interprétation par une machine acceptant directement le langage;

3. l'interprétation d'un codage interne de la requête, résultat d'une compilation partielle.

Bon nombre de systèmes supportant un langage dit de quatrième génération se sont tournés vers les optiques 2 et 3, notamment pour les avantages d'interactivité et de souplesse fournis par l'interprétation. Pour notre part, nous nous sommes tournés vers la première solution (Fig. 5.3.). Ce choix s'explique pour différentes raisons:

- La réalisation d'un interpréteur à la fois complet et efficace nous aurait demandé trop de temps et des connaissances que nous ne possédons pas encore (en assembleur notamment). L'usage d'un interpréteur dans des travaux effectués précédemment à l'Institut, a souvent mis en évidence des problèmes de performances.

- Nous pouvons supposer que les rapports demandés seront plutôt du genre volumineux et périodique. Dans ce cas, une compilation est plus intéressante qu'une interprétation, à condition que la compilation produise un code objet suffisamment performant, tâche peu aisée et de plus, propre à un jeu d'instructions machine.

  Dès lors, nous avons choisi de travailler en deux étapes pour la génération du code objet:

  * dans un premier temps, nous générons un code dans un langage de programmation (PLI),
  * dans un second, nous utilisons le compilateur de ce langage pour obtenir le code objet.

On peut critiquer ce choix du point de vue des performances. En effet, il donne lieu à une double analyse de la requête: une première fois par l'analyseur et une seconde par le compilateur. Si une requête doit passer par cette chaîne complète pour chaque exécution, c'est effectivement lourd, mais nous pouvons imaginer de stocker, soit le code PL/I généré, soit le code objet final.

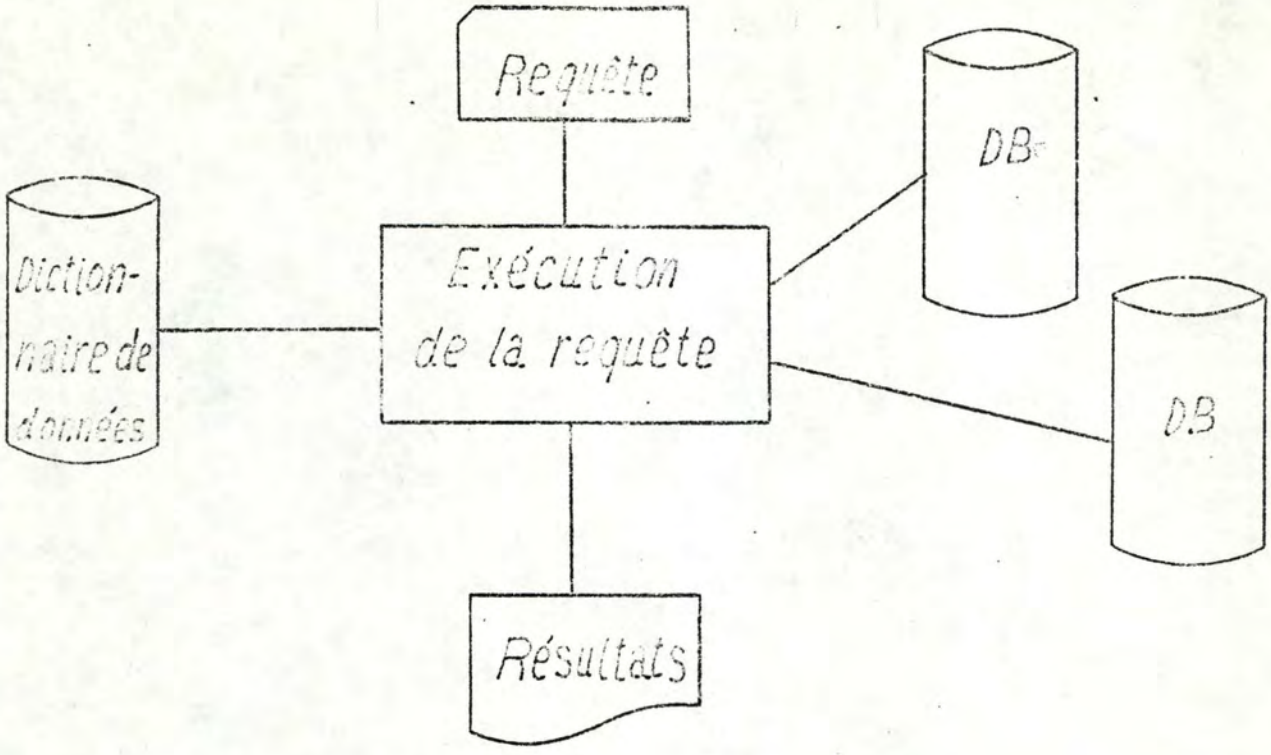


Fig. 5.2 : L'architecture de base

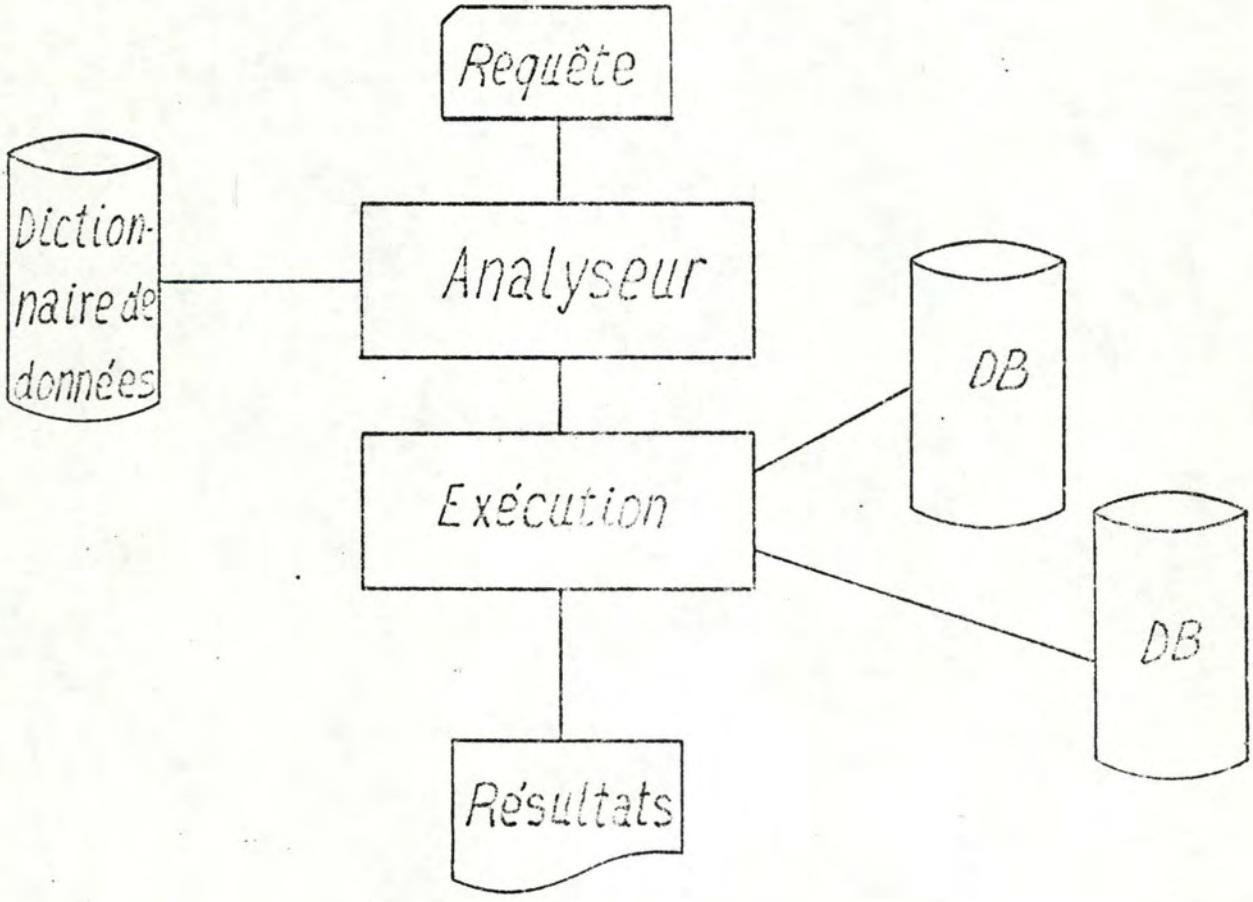


Fig. 5.3 : L'architecture en deux phases

- Ayant choisi d'utiliser un dictionnaire de données particulier comme méta-base de données (cfr. 5.3.3.), nous sommes obligés d'avoir des phases séparées pour l'analyse de la requête et pour l'exécution proprement dite. En effet, nous ne pouvons accéder à une autre base de données en même temps que nous effectuons l'accès au dictionnaire.

L'architecture d'exécution se transforme donc en un système de générateur de programmes où chacun d'eux se révèle être un générateur de rapport.

Par convention, nous réservons, sauf cas contraire, le terme 'générateur' pour désigner l'ensemble des deux opérations.

5.3.3. L'utilisation du DB/DC Dictionary d'IBM

Un point central du générateur est son dictionnaire de données. C'est en effet lui qui mémorise toutes les informations nécessaires à l'analyse et à l'exécution d'une requête. Dans notre cas, nous avons décidé d'utiliser un dictionnaire de données existant à l'ARBED, le DADIC d'IBM. Dans le chapitre 11, nous en verrons les caractéristiques générales. Maintenant, il nous reste à faire le choix d'une implémentation particulière. Trois possibilités s'offrent à nous:

Les SGBD's possèdent nécessairement une description des bases de données qu'ils manipulent. L'ensemble des descriptions des bases de données de tous les SGBD's cibles peut constituer le dictionnaire de données. Cependant, cette solution est peu satisfaisante, car chaque SGBD a souvent une description très personnelle, souvent dans un format illisible; un avantage cependant à souligner est le fait que l'information est toujours à jour.

Une autre solution consiste à utiliser un véritable dictionnaire de données.

Soit il en existe déjà un digne de ce nom. Si nous pouvons envisager de l'utiliser, cela représente une grosse économie de moyens.

Soit il n'en existe pas, nous nous décidons à en créer un. Dans cette dernière hypothèse, nous avons l'avantage d'être entièrement libres pour inclure tous les renseignements voulus.

Dans les deux cas, il faut maintenir la cohérence entre le dictionnaire de données et les descriptions des SGBD's.

5.3.4. L'utilisation du MAG

5.3.4.1. Le rôle du MAG

Pour être souple face aux différents SGBD's que peut exploiter notre générateur, nous avons choisi un langage d'extraction de données indépendant du modèle de données et des primitives utilisées par les SGBD's. De même, pour que l'exécution des requêtes soit indépendante, il faut qu'elle s'effectue dans le cadre d'une machine BD, elle aussi indépendante. Nous avons choisi d'utiliser comme base de réalisation le Modèle d'Accès Généralisé (MAG) pour le modèle commun et le jeu de primitives qu'il définit. Pour plus de détails sur ce modèle nous renvoyons le lecteur intéressé au chapitre approprié (chap. 3) ou aux références bibliographiques (H1) et (H2).

Vu que le MAG n'a pas d'existence propre, il doit se répercuter sur les SGBD's existants qu'il couvre. On peut imaginer deux types de transformations:

- soit on considère qu'il existe réellement à l'exécution, cette transformation se faisant par l'intermédiaire d'inter- faces simulant la machine MAG.

- soit on considère qu'il s'agit d'un outil théorique destiné à guider la conception du système vers une certaine généralité. Alors on ne lui donne pas d'existence réelle, car la transformation du MAG dans un modèle cible se fait lors de l'analyse.

Cette deuxième interprétation peut surtout être évoquée si on veut éviter des pertes de performances provoquées par l'utilisation d'interfaces.

Les SGBD's cibles de ces transformation se classent dans l'état actuel, dans trois catégories:

- IMS avec son modèle hiérarchique, qui contient les bases de données principales,
- DADIC avec son modèle entité-association, qui supporte la méta-base de données,
- un fourre-tout contenant les fichiers classiques accessibles à partir de PLI;

Pour chacun de ces modèles, il faut définir les règles de transformation.

Le problème que nous devons résoudre est de déterminer quand nous allons faire appel au MAG.

L'utilisation du MAG s'effectue différemment au niveau de l'analyse et au niveau de l'exécution d'une requête.

Pour pouvoir transformer la requête exprimée en LDA dans la machine cible MAG, il faut que l'analyseur accède à une description des bases de données, et de préférence une description exprimée dans le MAG. Le rôle du MAG, à ce moment, est d'être un outil de modélisation. De plus, la méta-base de données étant avant tout une base de données, il est normal qu'on y accède par une interface définie selon la convention du MAG.

Dans la phase d'exécution de la requête, nous retrouvons également ce second rôle.

Alors, nous obtenons l'architecture idéale, représentée par la figure 5.3.

Maintenant, nous devons vérifier s'il nous est possible de réaliser cette architecture en tenant compte des autres décisions déjà prises, à savoir l'utilisation du DADIC et d'IMS.

5.3.4.2. L'interface MAG - DADIC

Le choix ayant été fait d'utiliser le DADIC comme dictionnaire de données, deux problèmes restent à résoudre au niveau de la phase d'analyse:

1. Est-il possible de définir une interface entre la machine MAG et le DADIC.

2. Le DADIC permet-il d'exprimer le MAG comme modèle conceptuel de données.

Nous allons d'abord résoudre ce premier problème. Le DADIC est un SGBD travaillant sur des bases de données conçues suivant le modèle binaire d'entité-association. Les concepts de ce modèle ne sont pas nombreux. Nous y trouvons le concept d'entité appartenant à un type d'entité, d'attribut appartenant à un type d'attribut et d'association appartenant à un type d'association. La transformation vers le MAG est assez simple. Au concept d'entité, nous faisons correspondre celui d'article appartenant à un type d'article; de relation celui de chemin d'accès d'un type de chemin d'accès; d'attribut celui de valeur d'item d'un item.

Pour les primitives d'accès, la transformation est parallèle à celle des concepts. Ainsi, l'accès aux entités d'un type donne un accès aux articles d'un type, l'accès par une association se transforme en un accès dans un chemin et l'accès aux attributs devient l'accès aux valeurs d'item.

Cependant, une difficulté apparaît au niveau des attributs d'une association. Cette notion est tout-à-fait inconnue dans le modèle du MAG, qui ne possède, rappelons-le, que la notion d'item dépendant d'un article. A ce problème, nous proposons la solution classique de normalisation du schéma.

Nous créons dans le MAG pour chaque type d'association possédant un (des) attribut(s) un type d'article fictif du même nom que celui du type d'association. Et donc par un accès aux valeurs d'item de cet article, nous spécifions un accès aux valeurs d'attribut(s) de l'association.

Nous pouvons donc conclure que la création d'une interface entre le DADIC et la machine MAG est réalisable.

Nous allons maintenant nous intéresser au second problème, à savoir si le DADIC permet d'exprimer le MAG comme modèle conceptuel de données. Nous pouvons retrouver dans le DADIC la description de deux types de modèles de base de données.

1. La description du DADIC lui-même. Celle-ci s'exprime en termes de type d'entité, type d'association, type d'attribut et des relations entre type d'entité et type d'attribut, des relations entre type d'association et type d'attribut.

2. La description des bases de données gérées par IMS. Celle-ci s'exprime principalement par les types d'entité PSB, PCB, DB, SEGMENT, ELEMENT, ainsi que par des relations entre-eux. Si nous désirons décrire d'autres types de bases de données, il nous est possible de définir d'autres types d'entités et d'autres types de relations. Dans ces descriptions, beaucoup

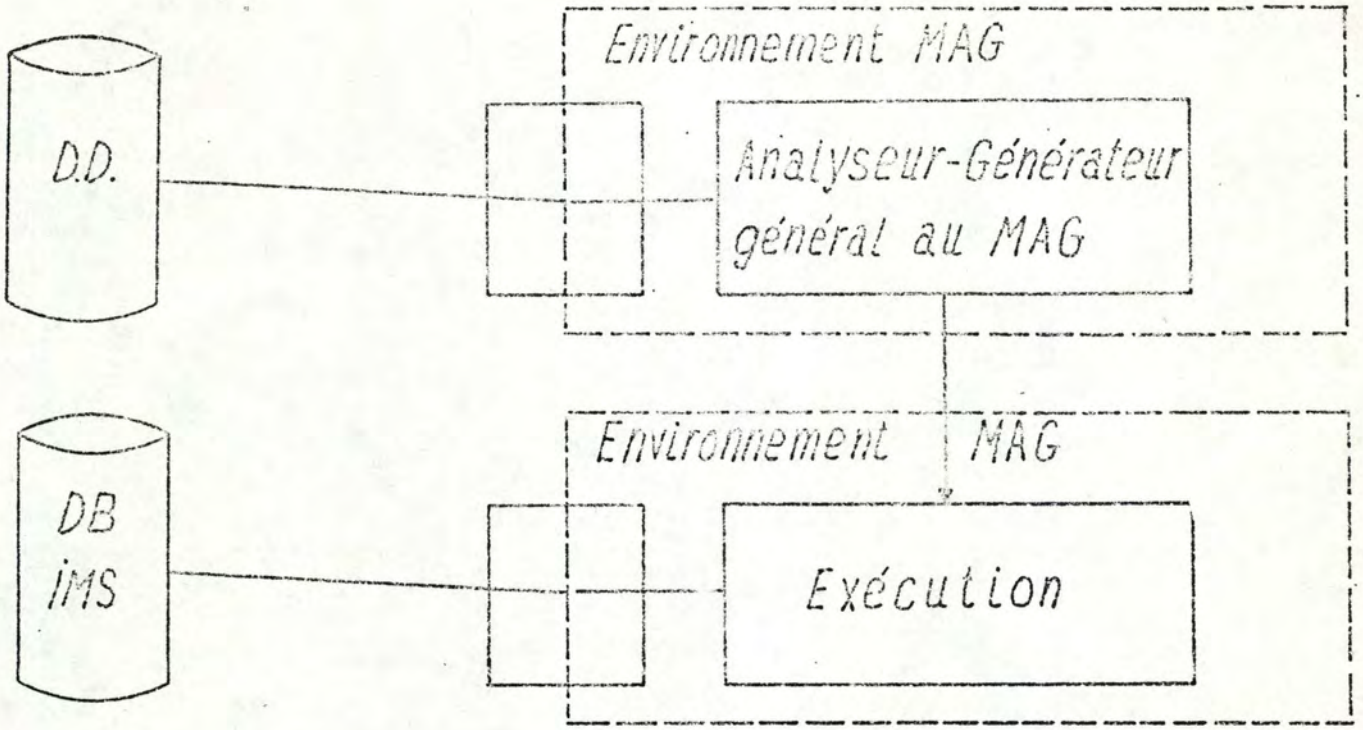


Fig. 5.4 : Le schéma général

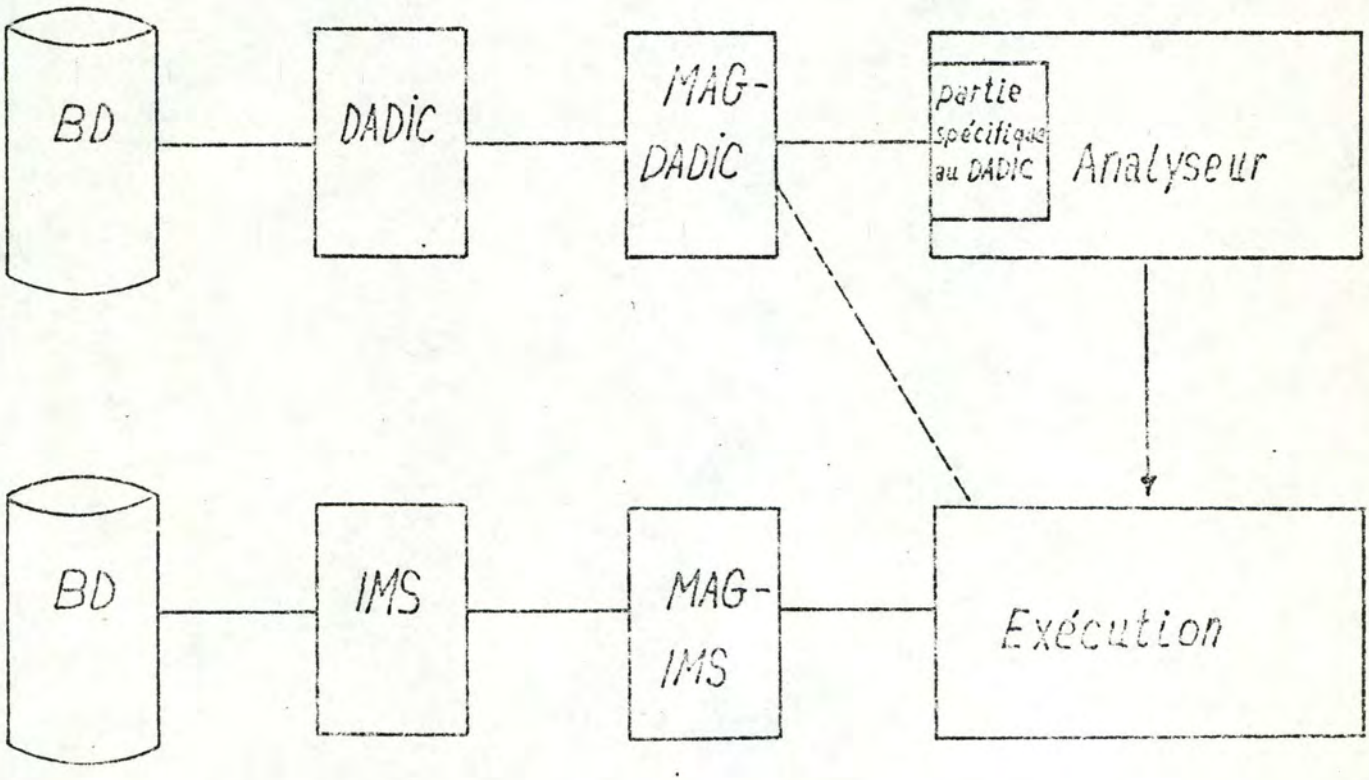


Fig. 5.5 : La réalisation

d'entités sont transformées par rapport au MAG en des attributs d'entités ou de relations. Dès lors, la transformation des concepts du modèle IMS dans le modèle MAG est assez lourde à mettre en oeuvre de manière automatique.

Par exemple, au type d'entité chemin d'accès du MAG peut correspondre l'existence d'entité de type association, s'il s'agit de chemin d'accès du dictionnaire de données. Par contre dans le contexte d'une BD IMS, il n'y a pas d'existence explicite des chemins d'accès. Il faut alors retrouver les attributs parent et enfant (logique ou physique) des relations de la base aux segments des articles source et cible du chemin.

A la seconde question, nous devons donc répondre par la négative. Cette réponse a pour conséquence que l'analyseur des chemins d'accès est, en partie, spécifique aux bases de données gérées par IMS.

5.3.4.3. L'interface MAG-IMS

La réalisation d'une interface entre IMS et MAG ne pose pas de problèmes insurmontables.

Les principaux concepts d'IMS trouvent leurs correspondants suivants dans le MAG:

PSB + PCB + DBD	base de données
type de segment	type d'article
segment	article
champ	item
valeur d'un champ	valeur d'item
champ de séquence	ordre dans un chemin
relation physique	chemin d'accès sans nom
relation logique	chemin d'accès dénommé par le type de segment effectuant le lien logique

Les primitives d'accès aux articles dans un chemin provoquent l'accès aux valeurs d'item et cachent des particularités d'IMS telles que l'usage des PCB's et la notion de positionnement.

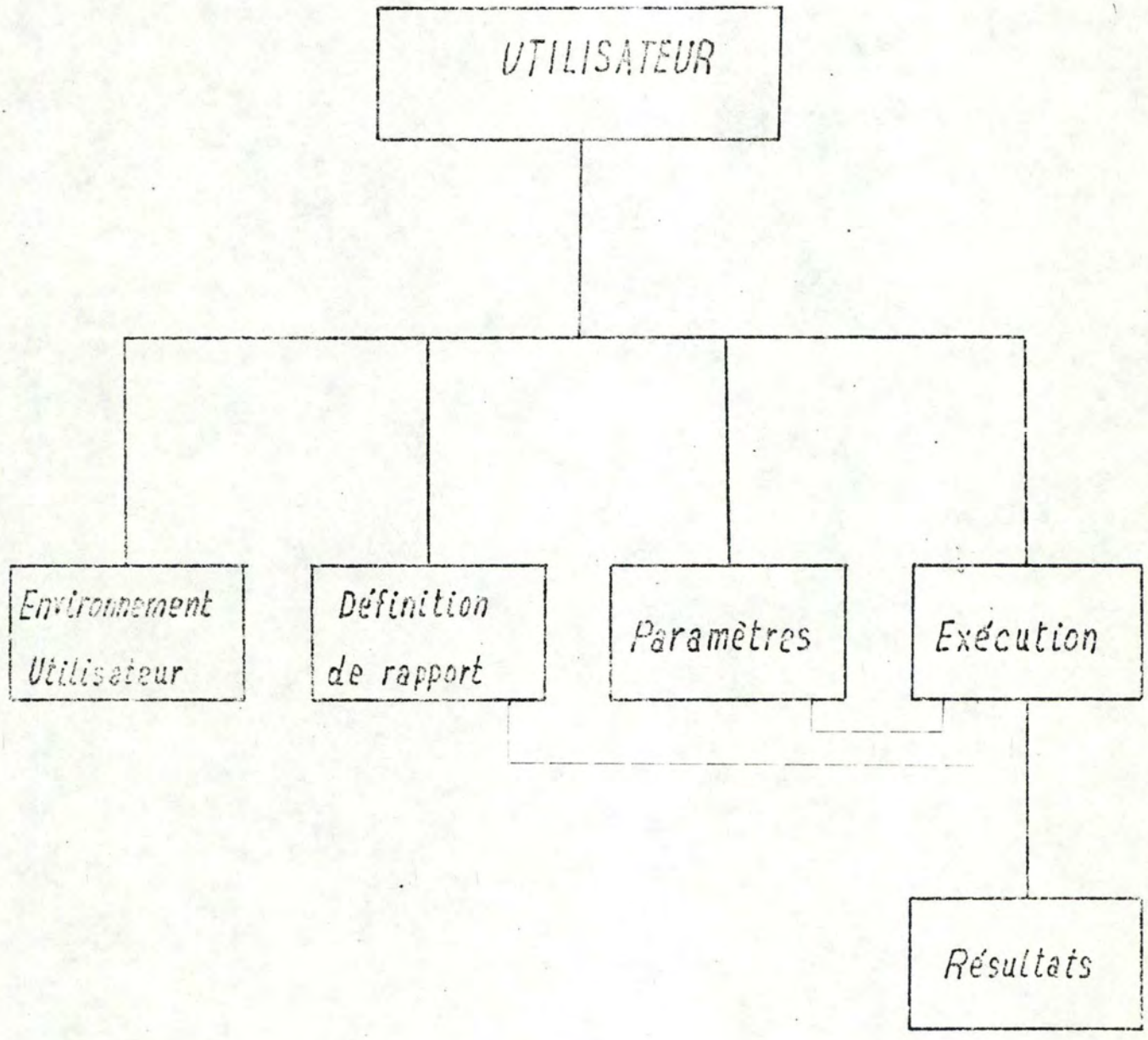


Fig. 5.6 : La Base de données du système

CHAPITRE 6: Le langage de définition logique

6.1. La double structuration d'un rapport

Pour obtenir son rapport, l'utilisateur doit introduire ses désidérata dans le système sous une certaine forme. Cette forme va servir d'interface entre le système et l'utilisateur. Elle est acceptable par l'utilisateur et assimilable par le système. Cette interface consiste en deux langages.

En effet, le rapport désiré présente une double structuration. Tout d'abord, il possède un certain format logique de sortie qui indique comment vont apparaître les informations recherchées sur le format de sortie. Ensuite, il suit un certain algorithme qui établit le contenu du rapport en termes de valeur d'informations.

Les deux structurations vont donner lieu à la définition d'un rapport en deux parties: d'un côté, la description logique grâce au langage de définition logique et de l'autre côté la description algorithmique grâce au langage d'extraction de données et de génération de rapports.

Il nous semble que ce type de définition présente les avantages suivants:

- une plus grande clarté pour l'utilisateur au prix, il est vrai, d'un effort de structuration de sa part dans l'analyse du rapport qu'il souhaite,

- la possibilité, pour nous, de traiter ces langages de manière plus logique. En effet, les interfaces entre les différents composants du système peuvent être définies de manière plus précise.

- la possibilité d'étendre proprement les possibilités des langages.

Dans ce chapitre, nous allons développer la description logique du rapport, tandis que nous consacrons le chapitre suivant à la description algorithmique par le langage d'extraction des données et de génération des rapports.

6.2. La description logique

La description peut s'effectuer au moyen d'un langage formel, d'une procédure interactive ou par procédés graphiques. Dans tous les cas, ils développent la notion de zone, concept de base de la description logique.

6.2.1. Le concept de zone

La zone peut être définie comme l'occupation d'une certaine surface sur le rapport de sortie du rapport. D'un point de vue géométrique, la zone peut prendre n'importe quelle forme (carré, rectangle, cercle, ...). Une des premières caractéristiques de la zone est qu'elle comprend ou non d'autres définitions de zones. Dans le cas où elle n'en comprend pas, on la qualifie de zone terminale ou de zone élémentaire, et elle correspond à une constante ou une variable dans la définition d'un rapport.

Une zone interne à une autre zone peut être générée plusieurs fois avant que la génération de la zone contenante soit poursuivie. Dans ce cas, la taille physique de la zone externe est ajustée automatiquement.

6.2.2. Les attributs d'une définition de zone

Nous allons présenter un ensemble non-exhaustif des attributs précisant chaque zone. Certains ne peuvent être utilisés que dans le cas où d'autres, précédemment déterminés, ont été mentionnés dans la définition.

Le premier attribut indispensable concerne la nomination de chacune des zones. Aussi, chaque zone possède deux noms. Le premier, le nom externe, est donné par l'utilisateur qui opte pour un identificateur qui parle de lui-même, de préférence. Ce nom est celui utilisé par l'utilisateur dans la description algorithmique à chaque fois que ce dernier veut référencer la zone. Le second, le nom interne, est produit automatiquement par le système et est utilisé lors de la gestion des zones pour désigner chacune d'elle.

Il consiste en une suite de trois nombres concaténés. Le premier reprend le niveau d'imbrication de la zone, le second le numéro de séquence local de la zone de niveau supérieur, le dernier le numéro de séquence local de la zone proprement dite dans ce niveau. La séquence d'apparition des zones d'un même niveau est définie de haut en bas, puis de gauche à droite.

Le second attribut essentiel est la position de la zone. Celle-ci est donnée pour un point caractéristique du type de la zone et par rapport au point supérieur gauche de la zone de niveau inférieur. Une même zone pouvant se répéter un certain nombre de fois, la notion de position est souvent relative et cet attribut est souvent mis-à-jour lors d'une exécution. La position doit être calculée dans les dimensions, aussi bien horizontale, que verticale.

L'utilisateur peut spécifier la répétitivité d'une zone. Ainsi, les attributs répétitivités minimale et maximale indiquent le nombre d'occurrences permis pour chacune des zones. Ainsi la répétitivité minimale peut être nulle si la zone n'est pas obligatoirement générée, égale à un si la zone doit être générée au moins une fois, égale à M si la zone doit au moins être générée M fois (répétitivité limitée), ou égale à '*' si la limite minimale n'est pas connue. Quant à la répétitivité maximale, elle est égale à un si la zone ne peut être générée qu'au maximum une fois, égale à M si la répétition est limitée à M fois ou égale à '*' si la répétitivité maximale n'est pas connue.

L'utilisateur doit indiquer comme attribut le type géométrique, car pour certains types des informations particulières sont requises ou prennent des significations différentes. Par exemple, pour la zone de type rectangle, la position doit être complétée par des informations sur la longueur et la hauteur relatives de celle-ci. Ces informations sont relatives, car on ne peut connaître avant l'exécution le nombre d'occurrences des zones internes. Dans le cas du cercle, en plus de la position du centre, il s'agit de connaître le rayon.

Pour personnaliser son rapport, l'utilisateur soigneux peut exiger qu'une zone soit soulignée ou encadrée. Ces deux notions ont une incidence sur la longueur et la hauteur de certaines zones.

Il est également possible de différencier une zone en- semble d'une zone terminale. Ceci se fait par l'attribut 'ROLE' qui identifie la zone ensemble d'une zone variable ou d'une zone constante. Cette différentiation provoque également l'enregistrement d'informations complémentaires, spécifiques à la zone. Si c'est une constante, il est nécessaire de connaître sa valeur et le centrage désiré (gauche / milieu / droit, haut / milieu / bas). Pour une variable, son format et son centrage ne sont pas inutiles.

Cette liste d'attributs n'est pas limitative et sera normalement sujette à de nombreuses extensions.

CHAPITRE 7: Langage d'Extraction de Données et de Génération de Rapports

7.1. Introduction

Le Langage d'Extraction de Données et de Génération de Rapports (LEDGR) a pour but de permettre la recherche d'informations dans des bases de données et de les présenter sous un format choisi par l'utilisateur. Langage de type procédural, il comprend à la fois des concepts de programmation classique, tels que variable, boucle, sélection, expression; des concepts propres aux DB, tels qu'instruction et boucle d'extraction de données; ainsi que des instructions liées à la génération de rapports.

Nous allons présenter dans l'ordre

- les objets manipulés par le LEDGR et leurs opérations primitives,
- des mécanismes de composition de ces algorithmes primitifs afin d'obtenir d'autres algorithmes spécifiques et complexes à volonté: expressions, instructions, procédures, etc.

Il est à noter que dans les descriptions de syntaxe, le '!' exprime un choix entre plusieurs possibilités.

7.2. Les objets du LEDGR

Les objets manipulés dans le LEDGR sont de 3 grandes classes:

- les objets de programmation classique (constante, variable, paramètre, ...).
- les objets propres aux BD's (segment, champ, ...).
- les objets propres aux rapports (zone, en-tête de page, fin de page, ...).

Les données peuvent provenir de plusieurs types de base suivants:

- Booléen: comprenant l'ensemble des valeurs de vérité (*vrai, faux*).

Les opérations de comparaison fournissent pour résultats des valeurs de ce type.

- Numérique: (entier et réel)
  Cela comprend respectivement l'ensemble des entiers (réels) positifs, négatifs ou nuls.
  Les opérations primitives fournies pour manipuler ces valeurs sont: addition, soustraction, négation, quotient par défaut, le reste par défaut, la multiplication, l'exponentiation, les différentes opérations de comparaison.

- Chaîne de caractères: Toute variable pouvant contenir une suite de caractères.
  Les opérations primitives applicables à ce type de base sont: concaténation, prendre une sous-chaîne, les opérations de comparaison, ...

Les objets de programmation classique

Les constantes sont de ces types de base.

Les variables peuvent être:

- scalaires ou,
- structurées:
  - en tableau: tous les composants y sont du même type de base. Le tableau peut posséder une dimension d'ordre supérieur à un. Les bornes de chaque indice sont statiques. Les éléments sont désignés par le nom du tableau indicé par une expres- sion entière pour chaque dimension, séparée par une ",".

```
Ex.: dcl tab1 (2:5, 1:10) : dec fixed;
     dcl tab2 (10)        : bin fixed;
     tab1 (3 * x + 7,4) = tab2 (ad);
```

- en record: les composants peuvent y être de type de base différent. Le record a un nombre fini de composants. Les éléments sont désignés par le nom du record, un point et leur nom propre. Il est possible de sauter la désignation du record dans la mesure où il n'y a pas d'ambiguïté.

```
Ex.: 1 compte,
          2 numéro: dec fixed (6)
          2 nom: char (30),
          2 adresse,
               3 rue: char (30),
               3 num: dec fixed (4),
          2 localité,
               3 code_post: dec fixed (5),
               3 commune: char (30),
          2 solde: dec fixed (6);
          .
          ...
```

Une valeur peut être en plus paramètre. Cela signifie qu'une valeur sera demandée à l'utilisateur lorsqu'il demandera une exécution du rapport, qui lui sera affectée avant de lancer l'exécution.

Les objets des bases de données

Une base de données est constituée d'une collection d'articles.

Chaque article est une unité d'information qui peut faire l'objet d'une création, d'une modification ou d'un accès. Chaque article est d'un type déterminé qui en définit les propriétés générales.

Aux articles d'un type peuvent être associées des valeurs d'items. L'item est défini comme un type de valeur, une valeur d'item étant une information se présentant comme une suite de symboles manipulable par un programme. Il est possible d'obtenir les valeurs d'item d'un article auquel on a accédé.

Un item peut être élémentaire (si aucun de ces fragment n'est significatif) ou décomposable (composé d'autre items).
Il peut être simple (une seule valeur par article) ou répétitif (plusieurs valeurs par article).

Les articles peuvent être structurés par des chemin d'accès inter-articles. Il s'agit d'un mécanisme qui associe à un article dit origine du chemin, des articles dit cibles, de manière telle qu'il soit possible, à partir de l'origine d'accéder dans un ordre déterminé aux cibles successives du chemin. Un chemin est unidirectionnel. Il appartient à un type qui en définit les propriétés générales.

L'accès aux articles se fait par l'intermédiaire de variables dites d'article. Des articles sont assignés à ces variables par l'intermédiaire de boucles d'accès ou par des ordres ponctuels. Les variables d'article ne doivent pas être déclarées. L'accès aux valeurs d'item d'un article se fait au travers de structures associées à la variable d'article, sauf cas particulier.

La correspondance dans le modèle IMS s'effectue de la façon suivante:

- base de données ----- les bases de données d'un PSB
- article ---------- segment
- item ---------- field
- chemin ---------- relation hiérarchique ou logique

7.3. Les symboles de base :

Tout programme LEDGR est défini comme une certaine suite finie de symboles de base. L'ensemble des symboles de base est défini comme suit:

<symbole de base> ::= <identificateur> ! <constante> ! <symbole réservé>

<identificateur> ::= <lettre> ! <identificateur><lettre> ! <identificateur><chiffre>

<lettre> ::= a!b!c!d!e!f!g!h!i!j!k!l!m!n!o!p!q!r!s!t!u!v! w!x!y!z!+!$!%!à! A!B!C!D!E!F!G!H!I!J!K!L!M!N!O!P!Q!R!S!T!U!V! W!X!Y!Z!

<chiffre> ::= 0!1!2!3!4!5!6!7!8!9

<constante> ::= <entier> ! <chaîne de caractères> ! <réel> ! <booléen>

<entier> ::= <chiffre> ! <entier><chiffre>

<réel> ::= <entier> . <entier>

<chaîne de caractères>:: = <chiffre> ! <lettre> ! <lettre><chaîne de caractère> ! <chiffre chaîne de caractères>

<booléen> ::= <les valeurs de vérité>

<les valeurs de vérité>:: = true ! false

<symboles réservés> ::= + - * ** / ( ) =) (: ; ^ ! & . /* */ ' " proc procédure returns return not if then else find for_each under over order_by on where grouped next zone until for while

null true false call dcl declare pic picture integer real float decimal dec fixed bin binary db database char footing_on footing_off heading_on heading_off character report end_report string varying by do end z_new z_afect z_end parameter database

7.4. Les mécanismes de composition

Nous allons présenter plusieurs de ces mécanismes:

Une instruction est une construction du langage qui, étant donné un ensemble de variables simples, de tableaux, de records, de variables de désignation de BD, de noms distincts et un ensemble de déclarations de fonctions et procédures de noms également distincts, spécifie une suite d'opérations à effectuer, susceptibles de modifier l'état de ces ensembles.

Une expression est une construction du langage, qui spécifie une suite d'opérations à effectuer pour calculer une valeur v.

Une expression de désignation peut être considérée comme une expression. Sa valeur sera la valeur de la variable qu'elle désigne. L'évaluation d'une constante produit toujours la même valeur quel que soit le contexte.

Si expr est une expression quelconque, la valeur de l'expression simple (expr) est la même que celle de expr (si celle-ci existe) et les évaluations de ces deux expressions se déroulent de la même manière.

Les déclarations de fonctions et de procédures sont des constructions du langage dont la sémantique peut être décrite approximativement comme suit:

- une déclaration de fonction spécifie un procédé de calcul d'une valeur v (valeur de la fonction) à partir d'un certain nombre d'autres valeurs v1, ..., vN (arguments de la fonction)

- une déclaration de procédure spécifie une suite d'opérations à effectuer sur un ensemble de variables et modifiant les valeurs de celles-ci.

7.5. Les instructions

On peut classer les instructions en trois grandes catégories:

- les instructions de programmation classique,
- les primitives d'extraction des BD's,
- les primitives de génération.

7.5.1. Les instructions de programmation classique

7.5.1.1. L'instruction composée

L'instruction composée spécifie que celles qui la composent, doivent être exécutées comme un tout. Elles sont délimitées par les symboles 'DO' et 'END' et séparées les unes des autres par le symbole ";".

Syntaxe:

<instruction composée>:: =
DO <liste d'instructions> END;

<liste d'instructions>:: =
<instruction simple>; !<liste d'instructions>
<instruction simple>;

Exemple:

```
DO
    i = i + 1;
    s = s + i * i;
    END;
```

7.5.1.2. L'instruction d'assignation et d'affectation

La plus fondamentale des instructions est l'instruction d'assignation. Elle spécifie que l'expression qui est à droite doit être évaluée et sa valeur affectée à la variable désignée dans sa partie gauche. Celle-ci peut être soit une variable de programmation, soit une zone terminale active.

Syntaxe

<instructions d'assignation>:: = <variable> =
<expression>!
<variable>:: = <identificateur>

Exemple

```
résultat =  a + b   * AIDA. QUANTITE. TOTAL1;
Z1 = résultat;
```

7.5.1.3. L'instruction conditionnelle

Cette instruction permet de choisir entre deux décisions en fonction de la valeur vraie ou fausse d'une condition.

Syntaxe:

<instruction if>:: = IF <expression booléenne>
THEN <instruction>
! IF <expression booléenne>
THEN <instruction>
ELSE <instruction>;

Exemples:

```
1) IF p1 ^= nul THEN p1 = father (p1);
2) IF x   15 THEN z = x + y ELSE z = 15;
```

7.5.1.4. Les instructions de répétition

Les instructions de répétition sont contrôlées par des conditions (de type 'TANT QUE' et 'JUSQU'A'), ou avec indice.

Dans le premier cas, on répète une action tant qu'une certaine condition est vérifiée. La condition est une expression logique (c'est-à-dire susceptible de prendre l'une des deux valeurs vrai et faux) dépendant des variables du programme. Une telle boucle n'est susceptible de se terminer que si l'action peut modifier l'un des éléments intervenant dans la condition ou si celle-ci est fausse dès l'origine.

a) l'instruction 'TANT QUE'

Une boucle de type 'tant que' est utilisée lorsqu'on désire atteindre un état du programme où une certaine condition, dépendant des variables du programme, est vraie. Si l'on connaît une certaine condition qui, intuitivement, "rapproche" l'état initial d'un état où la condition devient vraie, alors on utilise ce type d'instruction.

Syntaxe:

<instruction while>:: =
WHILE (<expression booléenne>)
<instruction>;

Exemple:

```
WHILE (i < 5)
  DO  a = a + x;
              i = i + 1;
          END;
```

b) l'instruction 'JUSQU'A'

Cette autre forme de répétition, voisine de la précédente, effectue une action et la répète jusqu'à ce que la condition soit vraie. A la différence du cas précédent, l'action est toujours exécutée au moins une fois, quelle que soit la valeur initiale de la condition.

Syntaxe:

```
<instruction repeat>:: =
     UNTIL  (<expression booléenne>)
          <instruction>;
```

Exemple:

```
UNTIL (a  b)
  DO b = (a 1 b)/2;
     a = (b - a)/2;
```

Dans le second cas, la répétition se réalise par un compteur sur un sous-ensemble des entiers, muni de l'ordre naturel. On peut y spécifier un pas au compteur.

c) L'instruction FOR

Syntaxe:

```
<instruction for>::=
     FOR
          <expression de désignation> =
          <expression arithmétique> TO
          <expression arithmétique> BY
          <expression arithmétique>
          <instruction>;
```

Exemple:

```
FOR i = 1 to 5 BY 2
     a(i) = a(i) + b(i);
```

7.5.1.5. L'instruction de saut ou de branchement

L'instruction de branchement ordonne de quitter l'exécution séquentielle normale des instructions pour la poursuivre à une instruction désignée par une étiquette.

Syntaxe:

<instruction de saut>:: = GOTO
<identificateur>;

Exemple:

```
DO
     IF i < 15 THEN z = x + y  ELSE GOTO fin;
i = i + 1;
          s = s + i * i;
fin : END;
```

7.5.1.5. L'instruction d'étiquette

L'instruction d'étiquette sert de référence à l'instruction de saut.

Syntaxe:

<identificateur>: <instruction>

Exemple:

```
DO
     IF i   15 TEN z = x + y  ELSE GOTO fin;
          i = i + 1;
          s = s + i * i;
fin:
     END;
```

7.5.1.6. Les commentaires

L'utilisateur pourra toujours rajouter ses commentaires à son programme LEDGR du moment que ceux-ci ne coupent pas un symbole de base. Ces commentaires seront traités comme une autre instruction, mais ils n'auront pas d'effet sur le contexte du programme (variables, fonctions, procédures).

Syntaxe:

<instruction de commentaire>:: = / * (texte libre) * /

Exemple:

```
FOR I = 1 to 5 STEP 2
DO a(i) = a(i) + b(i);
   b(i) = b(i) + 2;
END; /* Ceci est un commentaire:
        Fin de l'instruction FOR */
```

7.5.1.7. L'appel de procédure

Acceptant le concept de sous-programme, le LEDGR permet de faire appel à l'un d'eux de la manière suivante.

Syntaxe:

<appel de procédure>::= CALL <nom de procédure>
(<liste de paramètres effectifs>) !
<nom de procédure>
(<liste de paramètres effectifs>)
<liste de paramètres effectifs>:: = <expression arithmétique> !
<expression arithmétique>, <liste de paramètres effectifs>

Exemple:

```
CALL bidon (a, b, c, d * c);
```

7.5.2. Les instructions d'extraction

7.5.2.1. Présentation générale.

La boucle énumérative est une instruction qui demande l'exécution d'une instruction, appelée corps de la boucle, pour les éléments successifs d'une collection d'objets.

La forme générale se présente comme suit:

```
FOR_EACH <var> = <coll_ord_objet>
         <instr>;
```

où

<var> est le nom d'une variable du type des objets de <coll_ord_objet> ;
<coll_ord_objet> est l'expression d'une collection ordonnée d'objets;
<instr> est une instruction;

L'interprétation en est la suivante:

Pour chaque élément successif de la collection <coll_ord_obj>, on affecte cet objet (ou sa référence) à la variable ,<VAR> et on effectue l'instruction instr . La valeur de la variable <VAR> est indéfinie en dehors de la boucle.

Les objets de la collection peuvent être des articles, des collections d'articles ou des valeurs d'item. Nous allons détailler les boucles correspondantes dans les sections suivantes.

Afin de pouvoir se libérer de la contrainte formée par les boucles, il existe une instruction d'accès indépendante explicitée dans la section 7.5.2.5.

La définition d'expressions statistiques sera développée dans la section 7.5.2.6.

7.5.2.2. Boucle d'accès sur article

Dans cette boucle, les objets de la collection sont des articles.

* La syntaxe en est:

```
FOR_EACH  <VAR1> =  [<RANG>] <ARTICLE>
                    [WHERE     (<COND_ITEM>)          ]
                    [FROM      <VAR2>[VIA <CHEMIN> ]
                    [ORDER BY <LISTE_ORDRE>           ]
                    <INSTR>;
```

où

<VAR1> est un nom de variable d'article;
<RANG> est la désignation d'un intervalle de valeurs entières;
<ARTICLE> est le nom d'un type d'article;
<COND_ITEM> est une expression booléenne de

comparaisons dont la partie gauche désigne des items appartenant au type d'article de VAR1 ;
VAR2 est un nom de variable d'article ou de collection d'article
CHEMIN est le nom d'un chemin d'accès;
LISTE_ORDRE est une liste de désignation d'items appartenant au type d'article de ARTICLE . Chaque désignation peut être précédée de l'expression '(A)', pour ascendant ou '(D)' pour descendant. Sa valeur par défaut est '(A)'.

* L'interprétation en est la suivante:

- Les valeurs successives de la variable d'article VAR1 sont les éléments successifs de la collection d'articles définis par l'ensemble des clauses.

- La collection est constituée des articles de type d'article défini par ARTICLE .

- La sélection de certains articles peut être précisée par une expression booléenne portant sur des valeurs d'item de l'article. L'article n'est retenu que si l'expression est vérifiée.

- S'il n'y a pas de clause sur le chemin d'accès aux articles, l'accès se fait dans toute la base de données.

  Si la clause FROM est présente, elle spécifie

  - soit une variable d'article,
  - soit une variable de collection d'article.

Si elle est absente, ainsi que la clause de sélection, alors l'ordre est celui défini dans le schéma de la base de données pour ce type d'article dans le référentiel base de données ou chemin. Si elle est absente et qu'il existe une sélection, alors l'ordre est indéterminé.

- Si la clause <RANG> est présente, elle permet de ne sélectionner de l'ensemble des articles retenus par les autres clauses que ceux dont le rang est précisé dans <RANG>. Si elle est absente, l'accès se fait sur l'ensemble.

* L'accès à un article provoque normalement l'accès aux valeurs d'item des items qui lui sont rattachés. Celles-ci sont accessibles comme une variable structurée de nom <VAR1>. Cependant pour les items à la fois répétitifs et facultatifs, l'accès aux valeurs d'item ne se fait pas automatiquement et doit être géré par une boucle sur valeur d'item (cfr. supra).

7.5.2.3. Boucle d'accès sur collection d'articles

Dans cette boucle, les objets accédés sont eux-mêmes des collections d'articles.

* La syntaxe est:

```
FOR_EACH <VAR1> = [<RANG>] <ARTICLE>
                  [WHERE    (<COND_ITEM>)        ]
                  [FROM     <VAR2>[VIA <CHEMIN>]]
                  GROUPED BY <LISTE_ORDRE>
                       [HAVING (<COND_AGGR>) ]
                  <INSTR>;
```

où

<VAR1> est un nom de variable d'article;
<RANG> est la désignation d'un intervalle de valeurs entières;
<ARTICLE> est le nom d'un type d'article;
<COND ITEM> est une expression booléenne de comparaisons dont la partie gauche désigne des items appartenant au type d'article de <VAR1>;
<VAR2> est un nom de variable d'article ou de collection d'article
<CHEMIN> est le nom d'un chemin d'accès;
<LISTE_ORDRE> est une liste de désignation d'items appartenant au type d'article de<ARTICLE>. Chaque désignation peut être précédée de l'expression '(A)' pour ascendant, ou '(D)' pour descendant.
La valeur par défaut est '(A)'.
<COND_AGGR> est une expression de condition sur des fonctions statistiques.

* L'interprétation en est la suivante:

Les valeurs successives de la variable de collection d'articles <VAR1> sont les sous-ensembles, formés par le regroupement sur des valeurs d'item, des éléments de la collection d'articles définis par l'ensemble des clauses.

La boucle possède globalement la même signification que la boucle d'accès sur article, si ce n'est que la clause GROUPED, qui remplace la clause ORDER, spécifie un ordre sur les articles. Ainsi, ce procédé permet de faire des regroupements.

La clause HAVING permet de spécifier une expression booléenne de conditions. Celles-ci doivent porter sur des fonctions statistiques visant les valeurs d'item de chaque sous-ensemble. Si l'expression est vérifiée, le sous-ensemble est retenu.

* L'accès à une collection d'articles ne provoque pas l'accès aux valeurs d'item puisqu'une valeur d'item est rattachée à un article et non à une collection d'article. Pour accéder aux valeurs d'item, il est nécessaire d'utiliser préalablement une boucle d'accès sur article.

7.5.2.4. Boucle d'accès sur valeur d'item

* Dans cette boucle, les objets accédés sont des valeurs d'item.

* La syntaxe en est:

```
FOR_EACH <VAR1> =  [<RANG>] <ITEM>
                   FROM    <VAR2>
                   <INSTR>;
```

où

<VAR1> est un nom de variable d'item;
<RANG> est la désignation d'un intervalle de valeurs entières;
<ITEM> est le nom d'un item de l'article désigné par <VAR2>
<VAR2> est un nom de variable d'item;

* La boucle d'accès affecte successivement à la variable d'item de nom <VAR1> les valeurs d'item de nom <ITEM> rattachées à l'article référencé par la variable d'article <VAR2>. La collection peut être réduite par <RANG>.

7.5.2.5. Accès indépendant sur article

* La syntaxe en est:

```
FIND  [NEXT] <VAR1> = <ARTICLE>
      [WHERE   (<COND_ITEM>)    ]
      [FROM    <VAR2> [VIA <CHEMIN>]]
```

où

<VAR1> est un nom de variable d'article;
<ARTICLE> est le nom d'un type d'article;
<COND_ITEM> est une expression booléenne de comparaisons dont le membre de gauche désigne des items appartenant au type d'article de <VAR1>;
<VAR2> est un nom de variable d'article ou de collection d'article
<CHEMIN> est le nom d'un chemin d'accès;
<LISTE_ORDRE> est une liste de désignation d'items appartenant au type d'article de <ARTICLE>. Chaque désignation peut être précédée de l'expression '(A)' pour ascendant, ou '(D)' pour descendant. La valeur par défaut est (A)'.

* Cette instruction permet l'accès ponctuel à un article. Si la clause <NEXT> est absente, cela signifie que l'on désire accéder au premier article. Si la clause est présente, on désire le suivant de la collection d'articles définie d'après les mêmes règles que dans la section a.

A la différence des boucles, la variable d'une instruction d'accès ponctuel est définie dans tout le programme.

Si la clause <FROM> est présente <VAR2> doit être une variable d'article et non une variable de collection.
Les clauses <ORDER> et <RANG> ne peuvent être utilisées.

7.5.2.6. Les fonctions statistiques usuelles.

Le LEDGR contient des fonctions statistiques usuelles comme nombre, minimum, maximum, moyenne sur des valeurs d'item. Ces fonctions peuvent être appelées en tout endroit dans un programme LEDGR où une variable numérique peut se trouver.

* Syntaxe:

```
COUNT (<VARIABLE_ARTICLE1>[,
       <VARIABLE_ARTICLE2>])

AVG   ┐
TOTAL │ (<VARIABLE_ITEM>[,
      ┘ <VARIABLE_ARTICLE2>[)
MAXI  │
MINI  ┘
```

où

VARIABLE_ARTICLE1 et VARIABLE_ARTICLE2 sont des noms de variables d'article définies sur des boucles d'accès. Le calcul d'une expression statistique s'effectue sur des ensembles de valeurs d'item accédés à l'intérieur de boucles.

Il est nécessaire de spécifier deux informations:

1) la variable d'article et éventuellement l'item sur lequel doit se faire le calcul.

2) la variable permettant d'identifier la boucle qui définit l'ensemble des articles pour lesquels doit être calculée l'expression (contexte d'exécution).

La valeur d'une expression statistique n'est définie qu'en dehors de la boucle définissant son contexte d'exécution.

Si on ne spécifie pas <VARIABLE_ARTICLE2>, la boucle de contexte est la dernière boucle spécifiée dans le programme.

L'option COUNT signifie que l'on veut le nombre d'articles accédés. Elle doit donc spécifier un nom de variable. Les options AVG, TOTAL, MAXI et MINI portent sur des valeurs d'item et spécifient respectivement que l'on désire la moyenne, le total, le maximum et le minimum des valeurs d'items accédées dans la boucle de contexte.

7.5.3. Les instructions de génération.

Les opérations permises sur une zone sont:

- l'activation,
- la désactivation,
- l'assignation à une zone variable.

En plus, l'utilisateur a la possibilité de régler la pagination de son rapport.

7.5.3.1. L'activation

L'activation est la première opération à faire pour pouvoir travailler sur une zone. On ne peut activer une zone que dans la mesure où celle qui lui est supérieure hiérarchiquement est activée.

Syntaxe:

Z_NEW (<nom de zone>);

où <nom de zone> est un identificateur de zone de la description logique

7.5.3.2. La désactivation

La désactivation spécifie que l'on n'utilisera plus l'occurrence courante de la zone spécifiée. On doit désactiver une zone avant de pouvoir l'activer à nouveau.

Syntaxe:

Z_END (<nom de zone>);

7.5.3.3. L'instruction d'affectation

L'assignation est identique à l'instruction classique d'assignation, si ce n'est qu'elle désigne une zone terminale.
Cependant, sa syntaxe est différente.
Il est aussi important de signaler qu'on ne peut assigner une valeur à une zone terminale que dans la mesure où sa zone supérieure hiérarchique a été activée.

Syntaxe:

Z_AFFECT (<nom de zone>,<expr>)

où <expr> est une expression arithmétique.

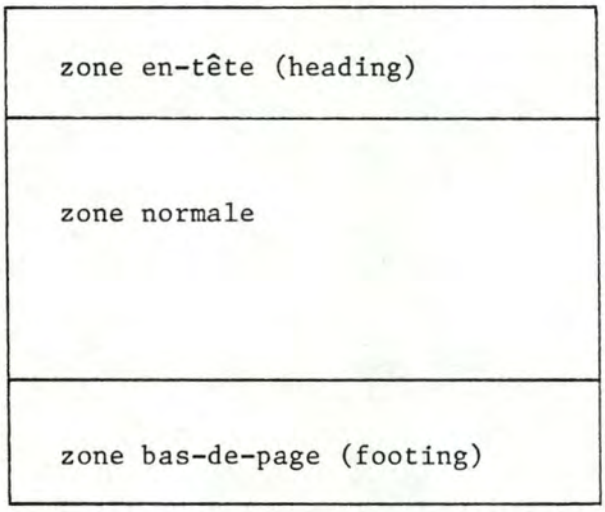
7.5.3.4. Les instructions de pagination

Le système de génération de rapports permet de gérer deux problèmes:

- celui de la numérotation des pages
- celui des en-têtes et bas de page.

La page est définie comme étant le support externe sur lequel sera visualisé le rapport. Elle peut être de tailles différentes suivant le choix de l'appareil de sortie (le format de la page de la liste, si le choix de l'utilisateur est porté sur une imprimante; le format de l'écran pour le choix du terminal).

Une page peut contenir une zone 'en-tête' et une zone 'bas-de-page' suivant le schéma ci-dessous:



Les zones 'en-tête' et 'bas-de-page' sont des zones-ensembles déclarées normalement grâce au LEDGR, mais elles ne peuvent cependant contenir d'autres zones-ensembles et doivent être déclarées comme des zones du plus haut niveau.

Dans la partie déclaration du LEDGR, on peut indiquer qu'une zone sert de footing (bas-de-page) ou de heading (en-tête). Par une série d'assignations, on peut y indiquer comment elles doivent être remplies. Ces assignations seront effectuées lors de la première écriture dans une page. C'est à l'intérieur de la zone normale que sont effectuées les instructions de génération de rapport.

L'utilisateur peut forcer explicitement la génération d'une nouvelle page grâce à l'instruction 'NEW_PAGE'. Il peut aussi spécifier dans le LDL qu'une zone possède l'attribut NEW_PAGE ce qui provoque un saut de page avant la génération de la zone.

Il peut provoquer le branchement ou le débranchement du système de pagination grâce aux instructions HEADING ON/OFF, FOOTING ON/OFF. Il peut aussi accéder à la variable qui contient le compteur de page: $PAGE.

7.6. Syntaxe des expressions

Il y aura toute opération définie comme opération primitive d'au moins un type de base.

Syntaxe générale:

<expression> ::= <expression simple> ! <expression arithmétique> !

7.6.1. Syntaxe des expressions simples

<expression simple> ::= <expression de désignation> !
<constante> ! <appel de fonction> ! (<expression>)

<appel de fonction>::= <nom de fonction>(<liste de paramètres effectifs>)
<liste de paramètres effectifs> ::= <expression arithmétique> ! <expression arithmétique> , <liste de paramètres effectifs>
<nom de fonction ::= identificateur>

7.6.2. Syntaxe des expressions arithmétiques:

<facteur>:: = <expression simple>
<terme>:: = <facteur> ! <terme> <opérateur multiplicatif> <facteur>
<opérateur multiplicatif>:: = * ! / ! **
<expression arithmétique>:: = <terme> !
<expression arithmétique> <opérateur additif> <terme> ! -<terme>

7.6.3. Syntaxe des expressions booléennes.

<proposition atomique>:: = <expression arithmétique> !
<expression arithmétique><opérateur relationnel>
<opérateur relationnel>::= =!^=!<!>!<=!>=!
<négation>::=<proposition atomique> ! NOT <proposition atomique>
<conjonction>::=<négation> ! <négation> AND <conjonction>
<expression booléenne>::=<conjonction> !
<conjonction> OR <expression booléenne>

7.6.4. Syntaxe des expressions de désignation

Une expression de désignation est une construction qui désigne UNE et UNE SEULE variable.

Syntaxe générale:

<expression de désignation>::=
<identificateur> (<couple d'expressions entières>) !
<identificateur> (<expression entière>) !
<liste d'identificateurs>
<liste d'identificateurs>::=<identificateur> !
<identificateur>.<liste d'identificateurs>

Un identificateur est une suite d'au plus 12 caractères non blancs parmi les lettres, les chiffres ou un blanc souligné (_) et commençant par une lettre.

Désignation pour les BD

Une BD est identifiée par un nom de BD.

L'identificateur d'un segment d'une BD est le nom de la BD, un point, un nom de segment. Le nom de segment est identificateur à l'intérieur d'une BD. Dans la mesure où le contexte permet de résoudre les ambiguïtés, on peut se limiter au nom de segment.

L'identificateur d'un champ d'un segment est l'identificateur du segment, un point, le nom du champ. Dans la mesure où le contexte permet de résoudre les ambiguïtés, on peut se limiter au nom de champ.

Les noms de BD, segment et champ doivent être répertoriés dans la BD de description des BD's accessibles (BD/DC data dictionary).

Syntaxe de désignation BD:

<expression de désignation BD>::=
    <nom de BD>.<nom de segment>.<structure d'un champ>!
    <nom de segment>.<structure d'un champ> !
    <structure d'un champ>
<structure d'un champ>::=<nom de champ> !
    <nom de champ>.<nom de sous-champ>

Exemple:

Soit la BD AIDA pour le segment NOM et le champ CODE/USINE

Désignations possibles:

pour la BD	:	AIDA
pour le segment	:	AIDA.NOM
		NOM -si pas d'ambiguïté-
pour le champ	:	AIDA.NOM.CODE/USINE
		NOM.CODE/USINE -si pas d'ambiguïté-
		CODE/USINE -si pas d'ambiguïté-

7.7. Syntaxe des déclarations de fonctions et de procédures

Si nous avons prévu la notion de fonction et de procédure, nous n'avons pas développé les concepts suffisamment pour y permettre l'emploi des instructions d'accès aux BD's, ainsi que celles de génération. Notre réflexion à ce sujet nous a permis de nous rendre compte que cet usage soulèverait de

nombreux problèmes non-triviaux (passage des paramètres qui seraient des désignations de zone, quel(s) PCB(s) utiliser pour exécuter cette procédure (fonction), ...) qui ne pourraient être analysés et résolus dans le cadre de mémoire sans négliger le reste de la solution. Nous nous contentons de soulever un voile sur ces problèmes en laissant la porte grande ouverte à une étude ultérieure.

Syntaxe

<déclaration de fonction>:: =<en-tête de fonction><bloc>
<déclaration de procédure>::=<en-tête de procédure><bloc>
<en-tête de fonction>::=
 PROCEDURE <identificateur> (<groupe d'arguments formels>)
  RETURNS (<type>) ;

<groupe d'arguments formels>::=<identificateur> !
  <identificateur>, <groupe d'arguments formels>

<type>:: = STRING ! REAL ! INTEGER ! <data type>

<data type>::= arithmetic variable ! string variable !

<arithmetic variable>:: = FLOAT ! FIXED ! BINARY !
     DECIMAL (<précision>)
<précision>::= entier , entier ! entier

Remarque:

la 'précision' est facultative, mais sera alors réglée par le compilateur PL/I.

<string variable>:: = BIT ! CHARACTER (<longueur>) VARYING

Remarque:

la clause 'longueur' est facultative. La clause 'VARYING' indique que la longueur sera variable.

<en-tête de procédure>::=
PROCEDURE <identificateur> (<groupe de paramètres formels>) ;

<groupe de paramètres formels>:: = <identificateur> ! <groupe de paramètres formels>, <identificateur>

<bloc>:: = <déclaration de toutes les variables> <déclarations des fonctions et procédures> <instruction composée>

<déclaration de toutes les variables>:: = <vide>!
DECLARE <déclaration de variables>; <déclaration de toutes les variables>!

DECLARE <déclaration de variables>;

Remarque:

Le ";" d'une fin de déclaration pourra être remplacé par une "," si celle-ci est suivie d'autres déclarations. Dans ce cas, on ne répétera pas le symbole réservé "DECLARE".

<déclaration de variables> :: = <déclaration de variables simples>!
<déclaration d'un tableau > ! <déclaration de record >

<déclaration de variables simples>:: = <identificateur>
<type>! (<liste d'identificateurs>) <type>

<déclaration d'un tableau>:: =
<identificateur> <déclaration d'indices> <type>!
<liste d'identificateurs> <déclaration d'indices> <type>

<déclaration d'indices > :: = <bornes d'indices>,
<déclaration d'indices>! <bornes d'indices>

<bornes d'indices>:: = <entier>:<entier> ! <entier>

<déclaration de record>:: =
01 <identificateur>, <liste de composants>

<liste de composants>:: =
<entier> <identificateur> <type>, <liste de composants>!
<entier> <identificateur> <type>!
<entier> <liste d'identificateurs> <type>!
<entier> (<liste d'identificateurs>) <type>, <liste de composants>

<déclaration des fonctions et des procédures>:: = <vide> !
<déclaration de fonction ou de procédure>; ! <déclaration des fonctions et des procédures>
<déclaration de fonction ou de procédure>;

<déclaration de fonction ou de procédure>::=
<déclaration de fonction> ! <déclaration de procédure>

<vide>:: =

Remarque:

la clause 'DECLARE' peut être abrégée en 'DCL',
la clause 'DECIMAL' peut être abrégée en 'DEC',
la clause 'BINARY ' peut être abrégée en 'BIN',
la clause 'CHARACTER' peut être abrégée en 'CHAR',
la clause 'PROCEDURE' peut être abrégée en 'PROC',

7.8. SYNTAXE DE LA DECLARATION D'UN PROGRAMME

Un programme est une construction dont l'exécution a pour effet d'interroger une ou plusieurs BD's pour en ressortir des informations qu'il transformera éventuellement pour les présenter selon un certain état sur un fichier de sortie.

Syntaxe:

<programme>:: = <identificateur> PROCEDURE;
<bloc programme>

<bloc programme>:: = <déclaration globale> <déclaration des fonctions et procédures> <instruction composée>

<déclaration globale>:: = DECLARE <type de déclaration>;
<déclaration globale> ! DECLARE <type de déclaration>;

Remarque:

Le ";" d'une fin de déclaration pourra être remplacé par une "," si celle-ci est suivie d'autres déclarations. Dans ce cas, on ne répétera pas le symbole réservé "DECLARE".

<type de déclaration>:: = <déclaration de BD accédée> !
<déclaration de variables> ! <description de définition logique> (<déclaration paramètre>)

<déclaration de BD accédée>:: = DATABASE ! (<nom BD>)

<description de définition logique> :: =
DECLARE REPORT <déclaration des zones> END_REPORT

<déclaration des zones> :: = <déclaration d'une zone> !
<déclaration d'une zone> <déclaration des zones>

<liste d'identificateurs>:: = <identificateur> !
(<identificateurs>)
<identificateurs>:: = <identificateur> !
<identificateur>, <identificateurs>

(<déclaration paramètre>) :: = PARAMETER (<déclaration de variables>)

Remarque:

la clause 'DATABASE' peut être abrégée en 'DB'.
la clause 'DECLARE' peut être abrégée en 'DCL'.

Chapitre 8: Le système de gestion de bases de données: Information Management System (I.M.S.)

8.1. Présentation générale

Une Base de Données (BD) est une collection de données élémentaires regroupées en segments qui seront reliés entre-eux par des relations hiérarchiques.

IMS fournit un haut degré d'indépendance par rapport à la structure de mémorisation. L'utilisateur ne doit quasi rien connaître quant à la structure physique du stockage des données.

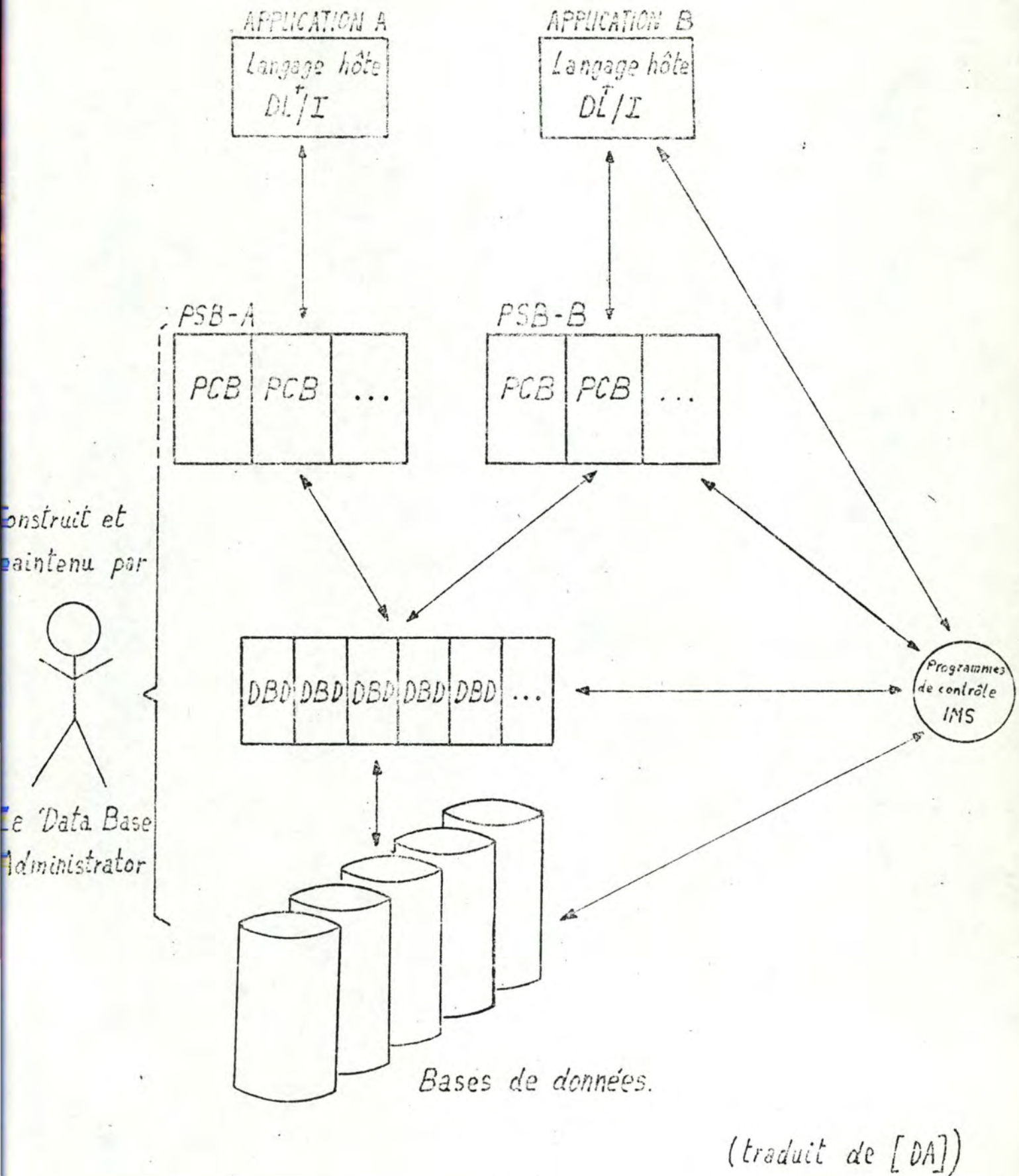
Chaque Base de Données (BD) est définie par un "Data Base Description" (DBD).

L'utilisateur n'opère pas directement au niveau des Bases de Données, mais plutôt sur la ou les vue(s) qu'il s'en fait. Chacune de celles-ci est définie au moyen d'un Program Communication Block (PCB). L'ensemble de tous les PCB's accessibles simultanément par un même utilisateur est appelé le Program Specification Block (PSB). L'architecture est illustrée par la figure 8.1.

En général, les utilisateurs sont des programmes d'application, écrits dans un langage hôte (PLI, Cobol, Assembleur 370) dans lequel le langage de manipulation de données est réalisé par des appels de sous-routines.

8.2. La structure de données d'IMS

En IMS, les types d'articles (types de segments) sont reliés entre-eux dans une structure arborescente. Les noeuds de la structure sont les types de segments. Le segment du niveau supérieur, unique, est nommé le segment racine. Les branches de la structure arborescente sont les types de chemin 1-N sans inverse et sans dénomination. La racine de l'arborescence ne



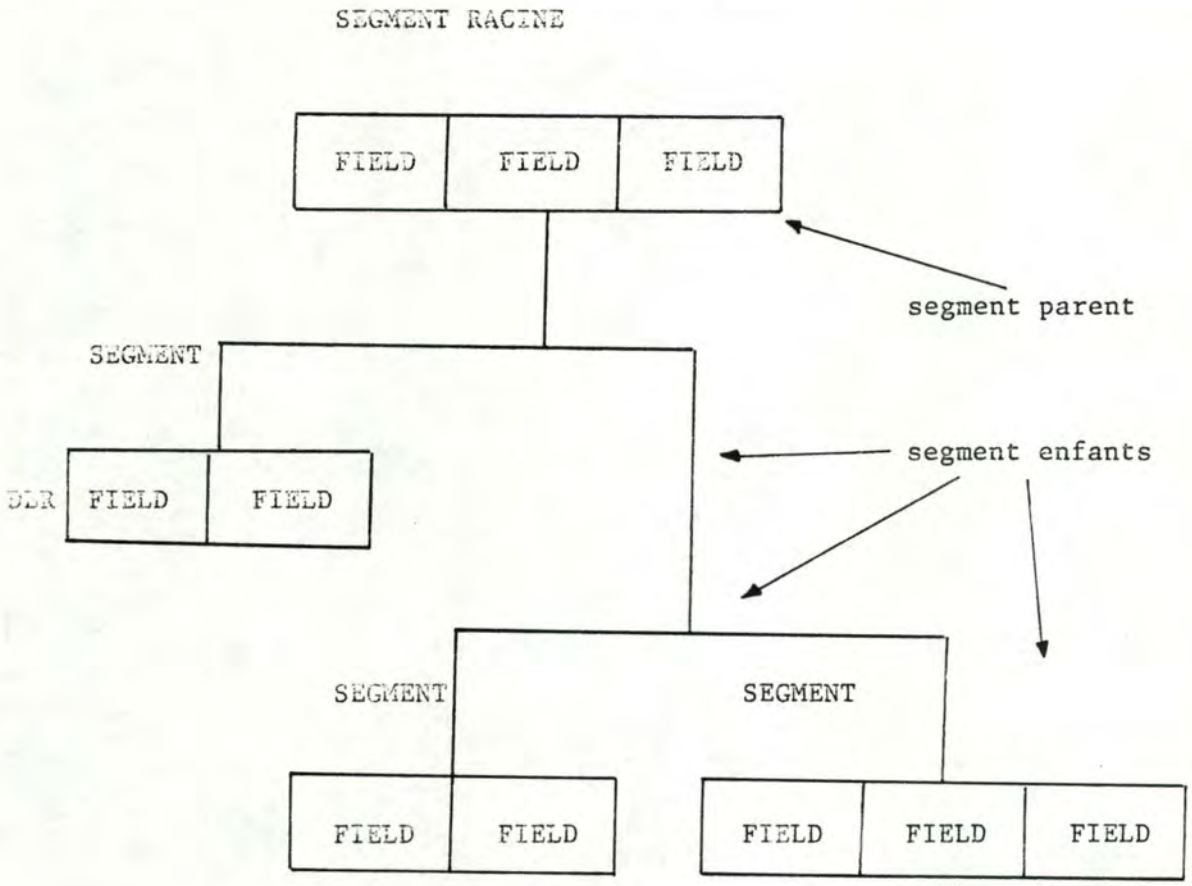
Construit et maintenu par

Le Data Base Administrator

(traduit de [DA])

Fig. 8.1: Architecture d'un système IMS.

descend d'aucun segment. Tout segment peut posséder 0,1 ou plusieurs segments descendants ou "enfant" et sera appelé segment parent par rapport au segment enfant. Aucune occurrence de segment enfant ne peut exister sans son parent. Ce dernier point est essentiel. Il signifie que la destruction de l'occurrence d'un segment donné, entraîne la destruction de ses enfants.



Du point de vue de l'organisation d'IMS, chaque BD est un ensemble ordonné d'éléments qui consiste en toutes les occurrences d'un "Database Record" (DBR). Un "Database Record" est l'ensemble formé par l'occurrence de la racine et toutes les occurrences qui dépendent d'elle. Chaque occurrence des types de segment peut être de longueur fixe ou variable. Le nombre de DBR's que contient une BD est égal au nombre d'occurrences de la racine.

Il est établi sur toutes les occurrences de tous les types de segment une séquence, appelée "SEQUENCE HIERARCHIQUE". Celle-ci se définit de haut en bas, puis de gauche à droite. Pour chaque segment, IMS définit une valeur de clé de séquence hiérarchique.

La plus petite unité de donnée qui puisse être accédée par une simple opération du langage de manipulation des données est un segment.

Le nombre maximal de niveaux est 15, tandis que le nombre maximal de segments dans une BD est 256.

Pour identifier et pour fournir un accès à un DBR particulier et à ses segments, DL/I définit des champs de séquence. Normalement, chaque segment dénote un champ comme le champ de séquence. Les champs de séquence doivent être uniques en valeur pour chaque occurrence d'un type de segment en dessous de son occurrence de parent. Ce champ n'est pas nécessairement défini pour chaque type de segment, mais est particulièrement important pour le segment racine. En effet, il sert d'identification pour le DBR. DL/I fournit un chemin d'accès direct rapide au segment racine du DBR, basé sur ce champ de séquence. Cet accès direct est étendu aux segments de niveau inférieur si les champs de séquence des segments le long du chemin hiérarchique sont spécifiés également.

La clé concaténée d'un segment consiste en la concaténation de toutes les clés depuis la racine jusqu'à celle du segment lui-même y compris et cela dans le chemin hiérarchique.

La base de données physique est composée d'un ou plusieurs fichiers dans lesquels se trouvent les segments.

8.3. Relations et bases de données logiques

Le langage de manipulation de données (DL/I) fournit une facilité pour mettre en relation les segments d'hiérarchies différentes. Cette facilité est présentée sous le nom de relation logique. Elle peut être définie comme toute liaison entre deux segments qui a une autre fonction que de réaliser la structure hiérarchique d'une base de données physique. Cette notion sera illustrée par la figure 8.2.

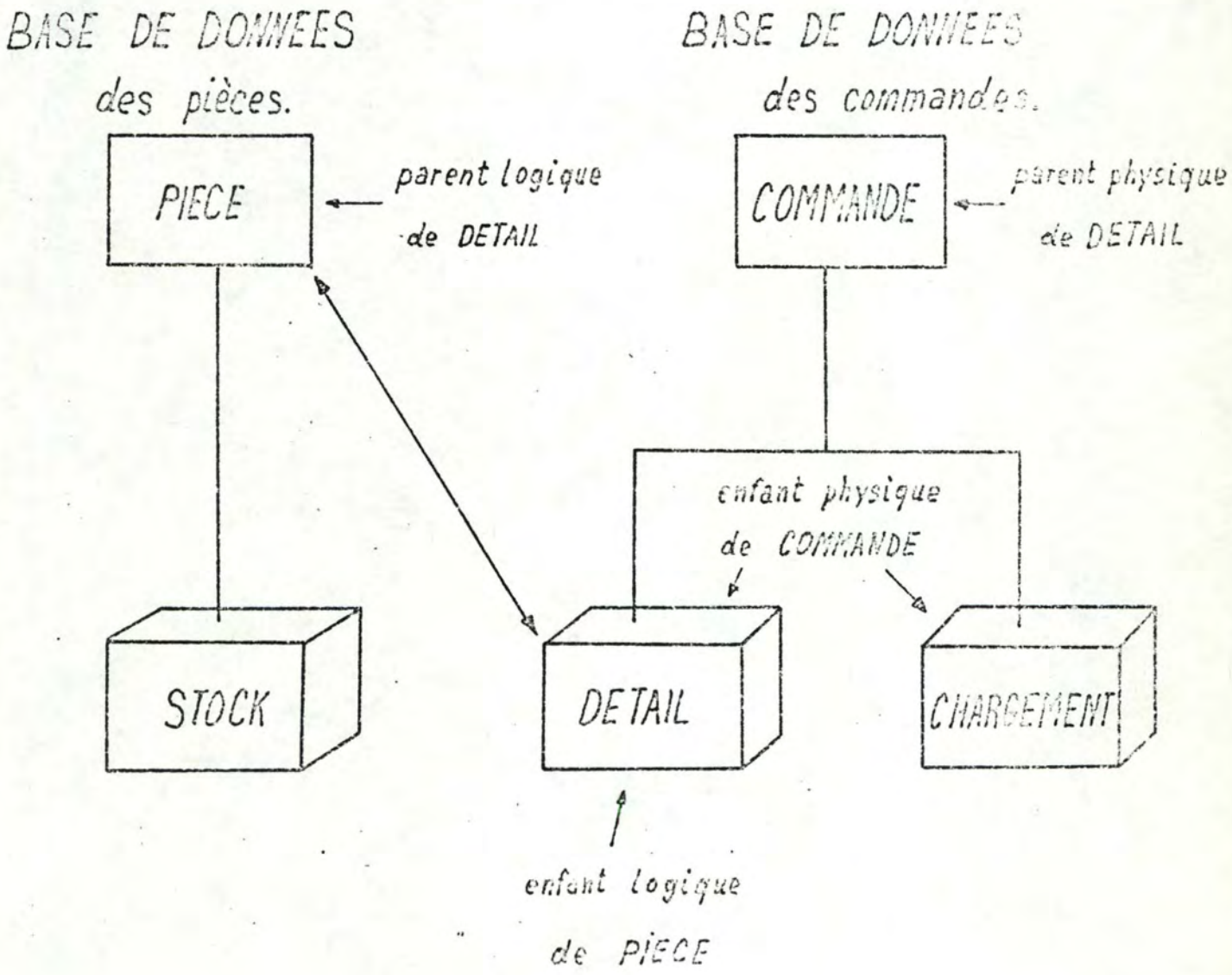
Par cette facilité, de nouvelles structures hiérarchiques sont définies. Elles fournissent des capacités supplémentaires d'accès aux segments désirés. Ces segments peuvent appartenir à la même base de données physique ou à des bases de données différentes. Une nouvelle base de données peut ainsi être définie et est appelée base de données logique.

Cette base de données logique permet la présentation d'une nouvelle structure hiérarchique au programme d'application, adaptée à la façon dont l'utilisateur voit les données à traiter. Cette adaptation peut se faire à deux niveaux:

- la LDB, telle qu'elle est vue, peut être composée uniquement de certains segments de la PDB,
- elle peut être composée, de plus, de segments appartenant à des PDB's différentes.

Le mécanisme de base utilisé pour construire une relation logique est de spécifier un segment dépendant comme un enfant logique, en le mettant en relation avec un second parent, le parent logique.

Dans la figure 8.2., le segment enfant logique DETAIL n'existe seulement qu'une fois, bien qu'il participe à deux hiérarchies. Il a un parent physique COMMANDE et un parent logique PIECE.



(tiré de [IBM2])

Fig. 8.2 : La relation logique

8.4. Le Data Base Description (DBD)

Chaque base de données (physique ou logique) est définie par un DBD.

1. Le DBD permet de décrire la structure d'une base de données.

Cette description se fera à l'aide de macro-instructions.

A chaque segment type correspond une "Macro-Instruction" SEGM. L'ordre des macro-instructions SEGM correspond à la séquence hiérarchique, mais l'énumération des segments selon la séquence hiérarchique ne peut à elle seule exprimer une structure et doit être complétée par un paramètre exprimant la position d'un segment par rapport aux autres. Le paramètre BYTES correspond à la longueur des données reçues ou fournies par un programme pour une occurrence du segment considéré, mais en aucun cas à celle de l'occurrence sur le support physique.

Chaque champ est défini par une macro-instruction "FIELD". Le début du champ est précisé par le paramètre "START" et sa longueur par le paramètre "BYTES". Cette longueur devra être respectée lors de l'écriture des critères de sélection (SSA).

Chaque segment peut avoir un champ privilégié que l'on appelle CLE du SEGMENT. Il est défini à l'aide du mot SEQuential.

2. Le DBD permet de déterminer les caractéristiques de l'organisation physique de la base de données considérée.

Pour déterminer le type de l'organisation physique, il faut pouvoir répondre à deux questions. Comment entrer dans la BD? Comment sont représentées les relations entre les occurrences d'un même "Data Base Record"?

Pour entrer dans la BD, deux méthodes sont disponibles:

- par un index,
- par une routine de type Direct Access Memory

Pour représenter les relations entre les occurrences d'un "DATA- BASE-RECORD", deux méthodes sont disponibles:
- par simple juxtaposition,
- par pointeurs.

La combinaison de ces différentes méthodes engendre différentes organisations:

INDEX	+	JUXTAPOSITION	=	HISAM
INDEX	+	POINTEURS	=	HIDAM
ROUTINE	+	POINTEURS	=	HDAM
		JUXTAPOSITION	=	HSAM

HISAM produit un accès indexé aux segments racines et un accès séquentiel aux segments dépendants.

HIDAM fournit un accès indexé aux segments racines et un accès par pointeur aux segments dépendants.

HDAM fournit un accès direct aux segments racines par une technique d'hashing et de chainage, avec un accès par pointeur aux segments dépendants.

L'organisation HSAM permet à la séquence hiérarchique d'être entièrement représentée par la contiguïté physique.

Lors de l'exécution d'un programme d'application, IMS consulte le DBD des bases de données physiques traitées de façon à pouvoir localiser les segments et tenir à jour les liaisons entre les segments.

8.5. Le Program Communication Block (PCB) et le Program Specification Block (PSB)

Chaque base de données logique est définie par un PCB. Dans celui-ci, nous retrouvons une spécification des correspondances entre la base de données logique et la base de données physique.

Le PCB est un bloc de contrôle dans lequel on indique les types de segment auxquels le programme a accès, de même que les traitements autorisés sur ces segments.

Comme pour un DBD, un PCB est écrit en utilisant des macro-instructions spéciales du langage assembleur que nous expliciterons infra.

Les PCB's d'un programme d'application forment ensemble le PSB. Ainsi, le PSB sélectionne pour le programme d'application les bases de données qui lui sont connues.

Lors de l'exécution d'un programme d'application, IMS consulte le PSB du programme pour pouvoir:

- ne fournir que les segments auxquels le programme a accès,
- refuser l'exécution d'ordres d'entrée/sortie non-autorisés.

Dans tous les cas, la situation se présente comme dans la figure 8.3.
Revenons à la définition des macro-instructions. Chaque macro-instruction PCB autorise l'emploi d'une base de données identifiée par le nom de son DBD.

Chaque macro-instruction PCB est suivie d'une ou plusieurs macro-instructions SENSEG dite de sensitivité. Chacune de celles-ci sélectionne un segment connu par le programme. Leur ordre est celui de la séquence hiérarchique. La sensitivité déterminée par ces macro-instructions doit respecter les chemins.

Le paramètre PARENT confirme la position du segment dans la structure.

La détermination des fonctions autorisées se fait au niveau du paramètre PROCOPT. Si ce paramètre suit la macro-instruction SENSEG, la fonction est

déterminée par rapport au segment. S'il est placé au niveau de la macro-instruction PCB, il concerne tous les segments référencés, sauf ceux dont la macro-instruction SENSEG contient elle-même un paramètre PROCOPT.

PROCOPT =			
	L	LOAD	Création initiale
	/		
	G	GET	Lecture
	I	INSERT	Ajout
	R	REPLACE	Modification
	D	DELETE	Suppression
	A	ALL (=GIRD)	Les 4 fonctions précédentes

Le paramètre KEYLEN du PCB représente la longueur de la plus longue clé concaténée qui puisse être construite.

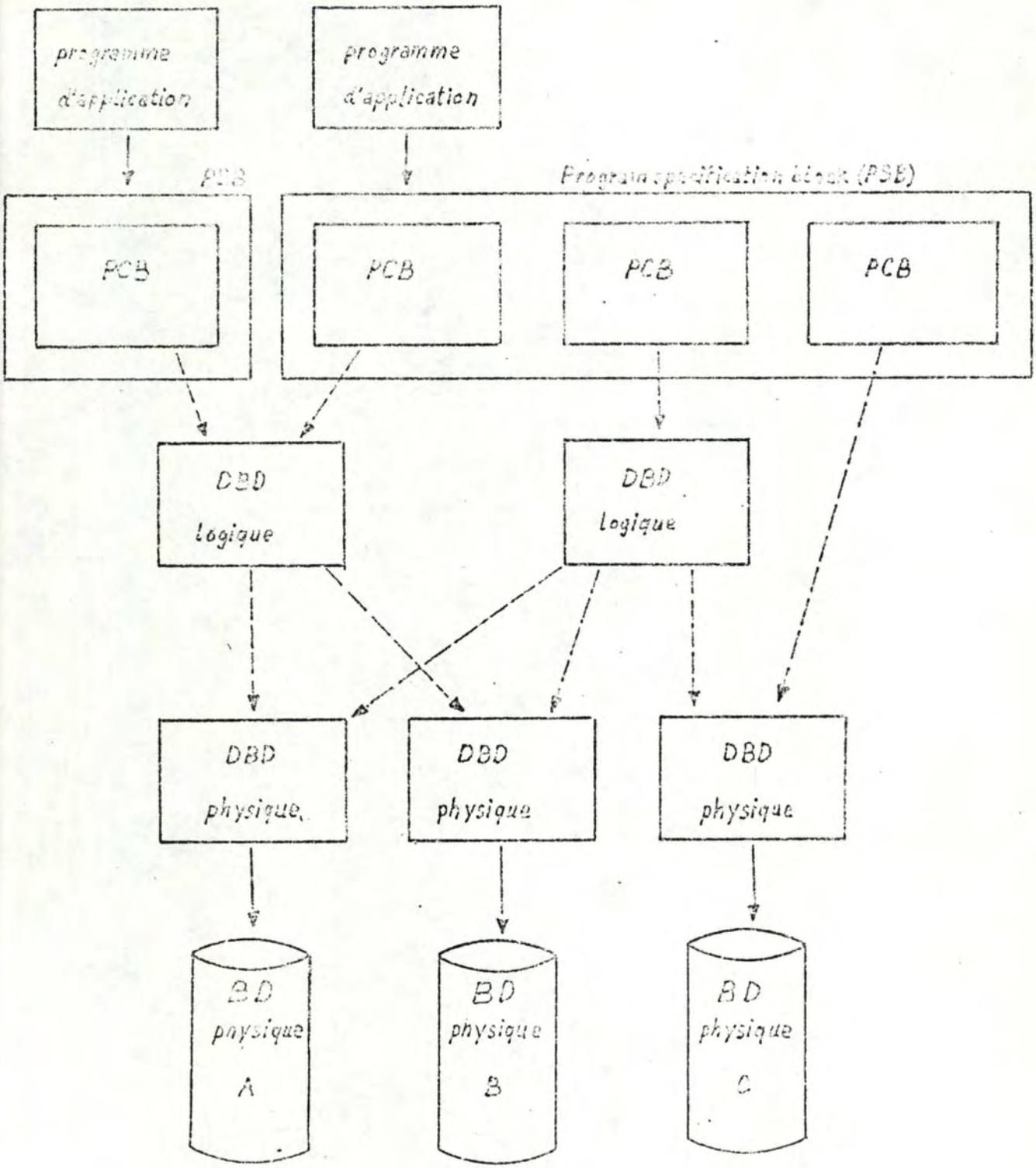
Le PSB est décrit par l'ensemble des descriptions des PCB's et se termine par une macro-instruction PSBGEN qui indique le nom du PSB et le langage utilisé pour écrire le programme qui sera contrôlé par ce PSB.

8.6. Le langage de manipulation de données: DL/I

Les "CALL's" d'un programme d'application permettent d'accéder aux segments d'une BD DL/I accessibles à celui-ci. Ces "CALL's" sont destinés à opérer une certaine fonction sur un segment ou sur un segment dans un chemin. Un "CALL" référence une liste de paramètres qui inclue toutes les données requises par DL/I pour compléter l'appel.

Les paramètres de celui-ci sont:

- l'adresse du PCB approprié,
- l'opération requise,
- l'adresse de la zone d'E/S,
- un ou plusieurs Segment Search Argument (SSA). Ceux-ci définissent les segments le long du chemin hiérarchique jusque et y compris le segment à traiter.



(tiré de [IBM2])

Fig. 8.3 : Accès à une base de données

1. Les différentes opérations

GU	Get Unique	accès direct
GN	Get Next	accès séquentiel
GNP	Get Next within Parent	accès séquentiel via le parent courant
GHU	Get Hold Unique	comme GU
GHN	Get Hold Next	comme GN
GHNP	Get Hold Next within Parent	comme GNP
		on utilise ces fonctions s'il est possible qu'on doive modifier ou supprimer le segment lu.
ISRT	Insert	pour ajouter un nouveau segment
DLET	Delete	pour supprimer un nouveau segment
REPL	Replace	pour modifier un segment existant

Le cheminement dans une base de données DL/I s'effectue de haut en bas et de gauche à droite. La position dans la base de données est celle du ou des segment(s) à partir duquel la recherche d'un autre segment commence. Normalement, DL/I retient la position de chaque niveau du chemin hiérarchique jusqu'au dernier segment accédé.

Get Unique: l'appel GU est utilisé pour accéder à un seg- ment spécifique ou à un chemin de segments d'une BD. Au même moment, il établit une position dans la BD à partir de laquelle des segments additionnels peuvent être traités dans une direction vers l'avant.

Get Next: L'appel GN est utilisé pour retrouver le segment suivant ou un chemin de segments dans une BD. Normalement, cet appel se déplace vers l'avant dans la hiérarchie d'une BD depuis la position courante. Il peut être forcé à commencer à une position précédente par un code de commande, mais sa fonction normale est de mouvoir vers l'avant depuis un segment donné jusqu'au prochain segment dans la BD.

la forme HOLD des appels GET: un GHU, ou GHN indique l'intention de l'utilisateur d'exécuter un appel "DELETE" ou "REPLACE".

INSERT: Cet appel est utilisé pour insérer un segment ou un segment dans un chemin dans la BD. Il est utilisé aussi bien pour le chargement initial des segments dans la BD, que pour ajouter des segments dans les BD's existantes.

Pour contrôler où les occurrences d'un type de segment sont insérées, l'utilisateur définit normalement un champ unique de séquence dans chaque segment. Quand celui-ci est défini dans un type de segment racine, le champ de séquence de chaque occurrence du type de segment racine doit contenir une valeur unique. S'il est défini pour un type de segment dépendant, le champ de séquence de chaque occurrence en dessous d'un parent physique donné, doit contenir une valeur unique. Si aucun champ de séquence n'est défini, une nouvelle occurrence est insérée après la dernière existante.

DELETE: Cet appel est utilisé pour détruire un segment d'une BD. Si un segment est détruit d'une base de données DL/I, tous ses dépendants le sont également.

REPLACE: Ce dernier sert à modifier les données dans une partie des données d'un segment ou d'un segment dans un chemin d'une BD. Les champs de séquence ne peuvent pas changer avec cet ordre.

2. Le(s) Segment Search Argument (SSA)

A l'aide de SSA, on peut demander un segment qui répond à des conditions déterminées.

Découpe du SSA: un SSA contient trois parties. Au minimum, il contient le nom du type de segment. Optionnellement, il peut contenir également des codes de commande et/ou des qualifica- tions de champ.

nom du segment	codes commande	qualification des champs

Un SSA peut contenir 8 statements de qualification maximum reliés par des opérateurs booléens. Un statement de qualifica- tion peut être absent. La valeur donnée dans le statement de qualification est comparée au contenu d'une zone de segment.

Les codes de commande sont une extension de la fonction CALL. En principe, ils n'ajoutent pas de nouvelles possibilités, mais ils facilitent le travail du programmeur. Ils spécifient une variation fonctionnelle du "call" telle que retirer la dernière occurrence du segment en dessous de son parent.

Dans un statement CALL, il peut y avoir un SSA pour chaque niveau de la structure hiérarchique.

Chaque qualification de champ contient un nom de champ, un opérateur relationnel et une valeur de comparaison. Quand des occurrences du type de segment sont recherchées par DL/I, le champ spécifié est comparé à la valeur de comparaison comme l'opération relationnelle le spécifie.

Si seulement le nom du type de segment est spécifié, la première occurrence rencontrée de ce type satisfaira le "call".

8.7. Principes de la programmation DL1.

En OS, une procédure cataloguée génère un certain nombre de cartes de contrôle. Notons immédiatement que les figures sont tirées de [IBM4].

1. (Fig. 8.4.). La carte EXEC générée provoque le chargement par le système d'un ensemble de modules et de blocs de contrôles, appelés DL1.

2. (Fig. 8.5.) DL1, qui se trouve maintenant en mémoire, analyse les indications transmises par les cartes de contrôle, à commencer par le nom du PSB. Il s'agit en l'occurrence de PSB1 que DL1 fait charger en mémoire par l'OS.

3. (Fig. 8.6.) DL1 analyse le PSB. Comme ce PSB contient deux PCB's et qu'ils renvoient respectivement aux DBD's DBD1 et DBD7, DL1 fait charger ces deux nouveaux blocs de contrôle par l'OS. Il va procéder maintenant à leur rapprochement et vérifier entre autres que les macro-instructions SENSEG de chaque PCB correspondent bien aux macro-instructions SEGM du DBD référencé. Le respect des règles de

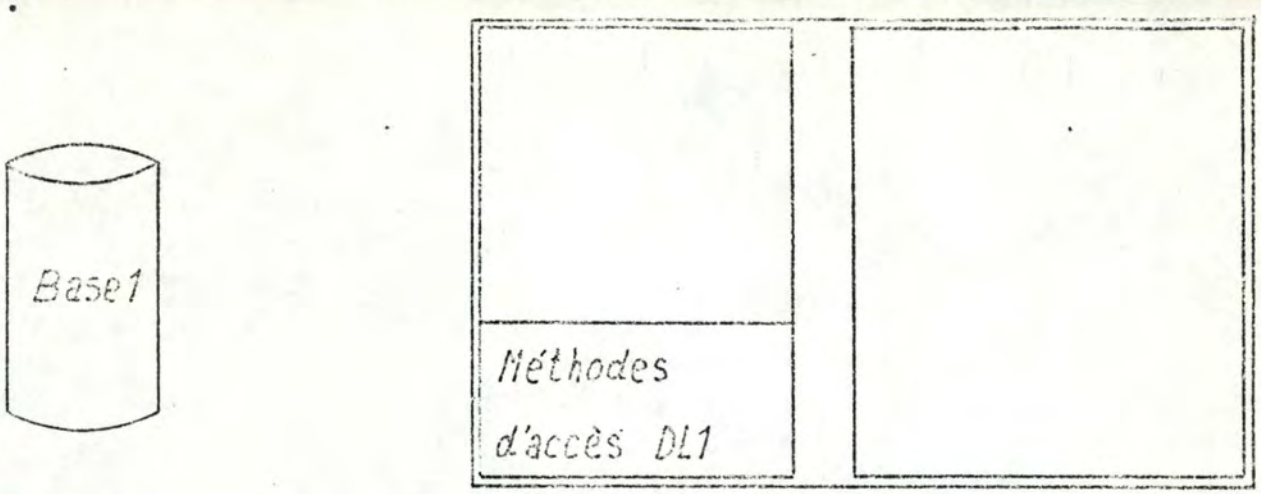


Fig. 8.4 : Chargement de DL1 en mémoire

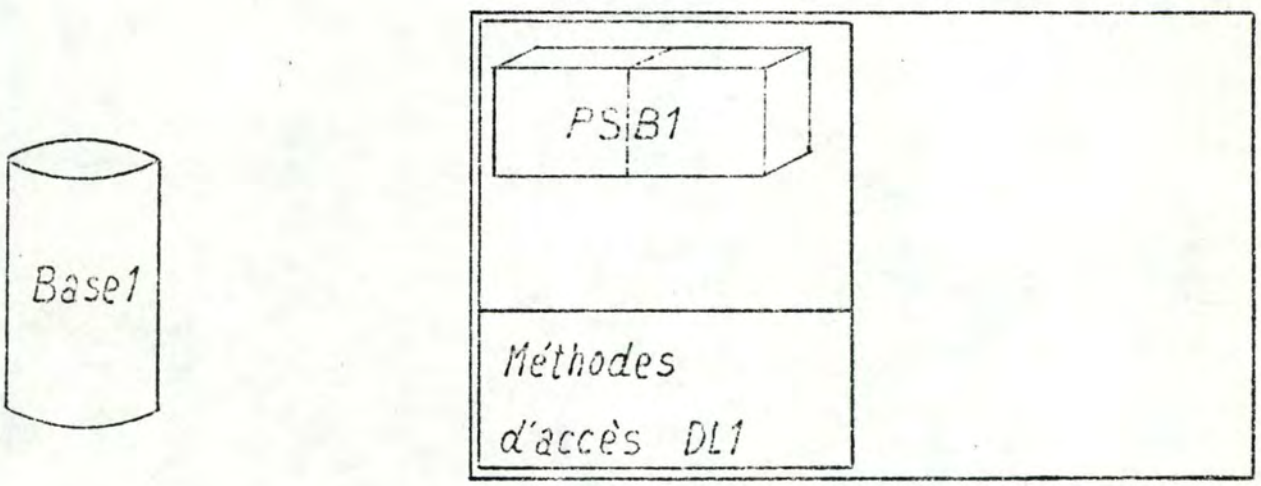


Fig. 8.5 : Chargement du PSB par DL1

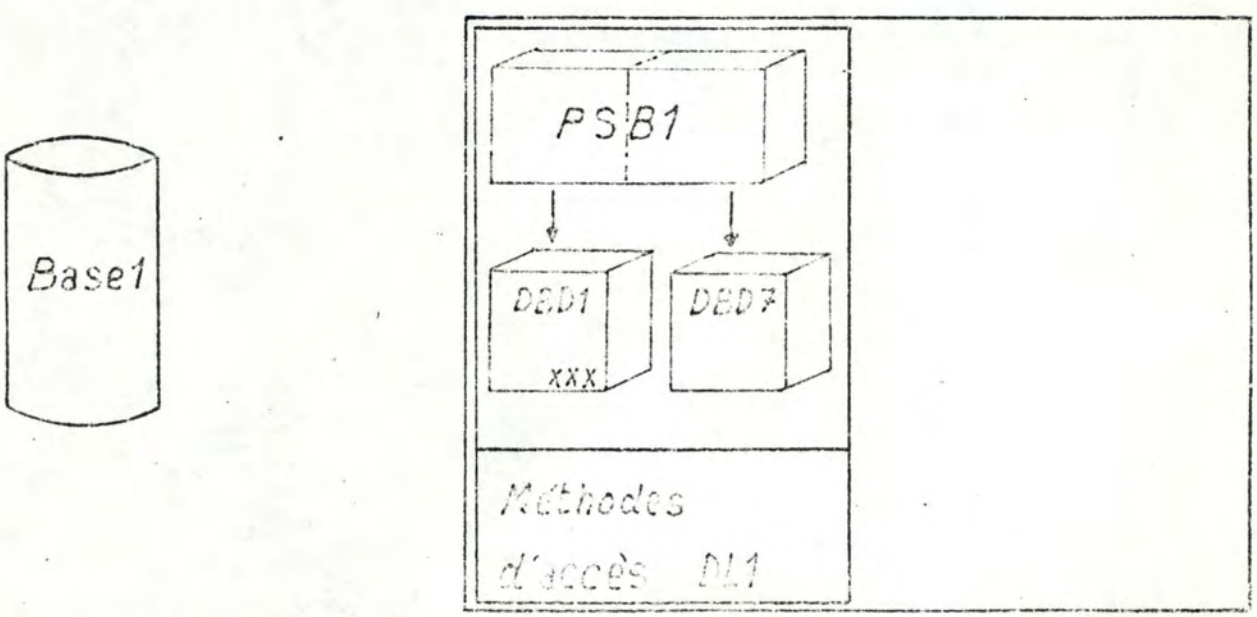


Fig. 8.6 : Analyse du PSB, Chargement des DBD's par DL1

sensitivité tel que commencer un chemin par la racine, l'interdiction de sauter un segment dans un chemin est soigneusement vérifiée. DL1 s'assure aussi qu'une place est réservée pour contenir la plus longue des clés concaténées. Cette troisième étape peut être la dernière, car toute irrégularité constatée lors du rapprochement DBD / PSB provoque une fin anormale accompagnée d'un code indiquant la raison.

4. (Fig. 8.7.) Supposons qu'aucune irrégularité n'ait été décelée, DL1 peut donc faire charger le programme utilisateur (programme d'applciation) dans la même région ou partition. Ce programme recoit deux adresses. En effet, le PSB qui le contrôle comporte deux PCB's. L'ordre dans lequel sont reçus ces adresses correspond à l'ordre d'écriture des macro-instructions PCB dans le PSB. Il n'est pas impossible de retrouver un même DBD référencé par plusieurs PCB's. Les différentes adresses correspondront à la même BD, mais avec une sensitivité différente et probablement des PROCOPT différents.

5. (Fig. 8.8.) Un programme DL1 a pour objectif le traitement d'une ou plusieurs BD's. Chaque opération à exécuter sur l'une d'entre elles est demandée à DLI par un ordre CALL. Trois paramètres au moins l'accompagne:

   - tout d'abord une zone contenant le code fonction,
   - l'indication de la BD à traiter, en donnant une des adresses précédemment reçues, et
   - l'adresse d'une zone d'entrée / sortie.

   Le programme utilisateur ne cède pas la main à DL1, mais à cette interface ajoutée au programme utilisateur. Cette interface est conçue pour homgénéiser les demandes exprimées par des CALL's provenant de différents langages (Assembleur, Cobol ou PL/I).

6. (Fig. 8.9.) Cette interface donne à son tour le contrôle à DL1 qui va analyser les différents paramètres associés à l'ordre CALL. S'il s'y trouve une erreur, cela peut entrainer un simple code retour qui n'interrompt pas le programme (comme par exemple, une fonction inconnue ou non prévue par le PROCOPT). Mais, si par exemple l'adresse de PCB n'est pas l'une de celles reçues par le programme, DL1 lance un ordre d'entrée/sortie en direction de l'unité contenant la BD désignée.

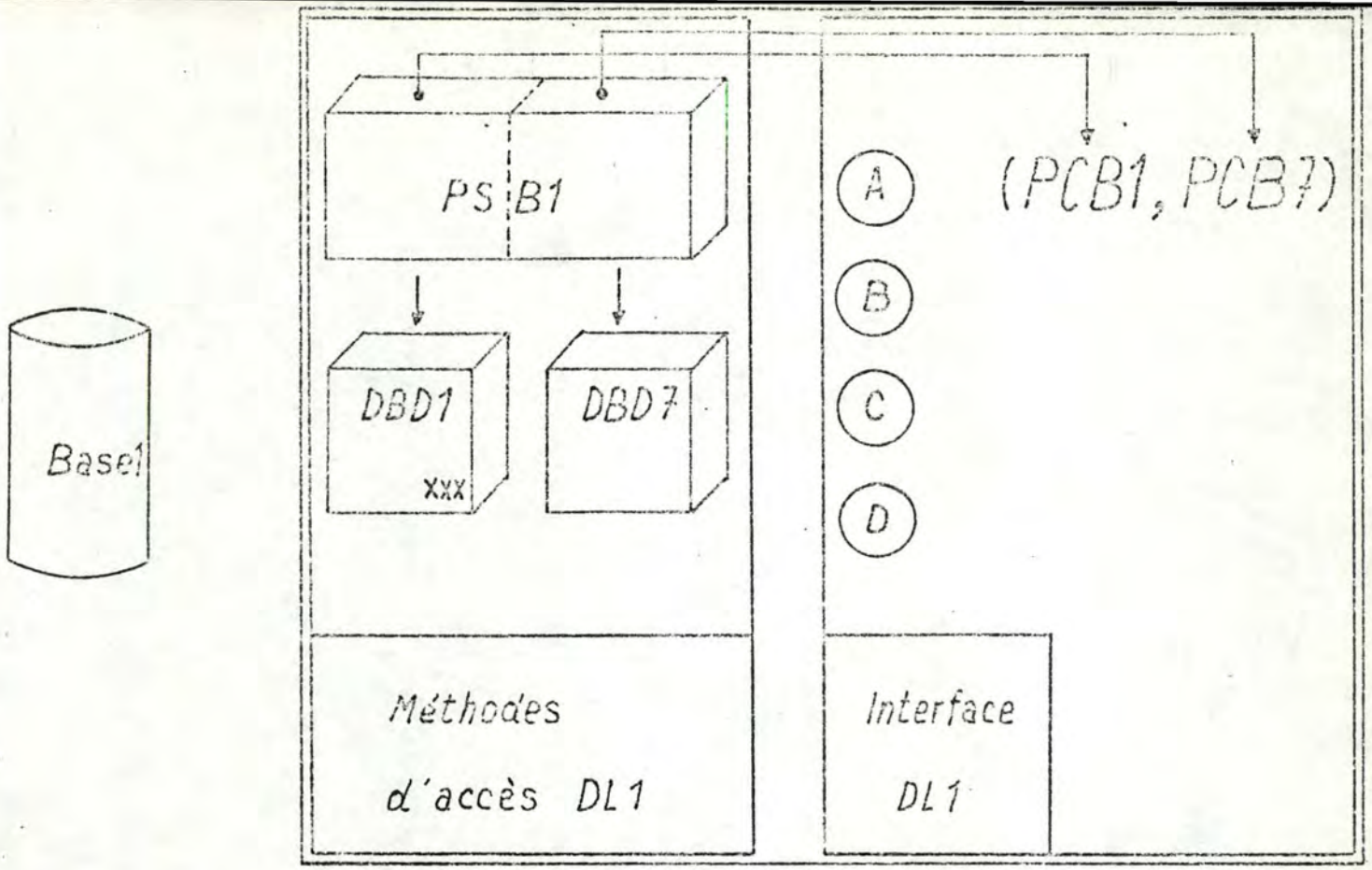


Fig. 8.7 : Chargement du programme utilisateur

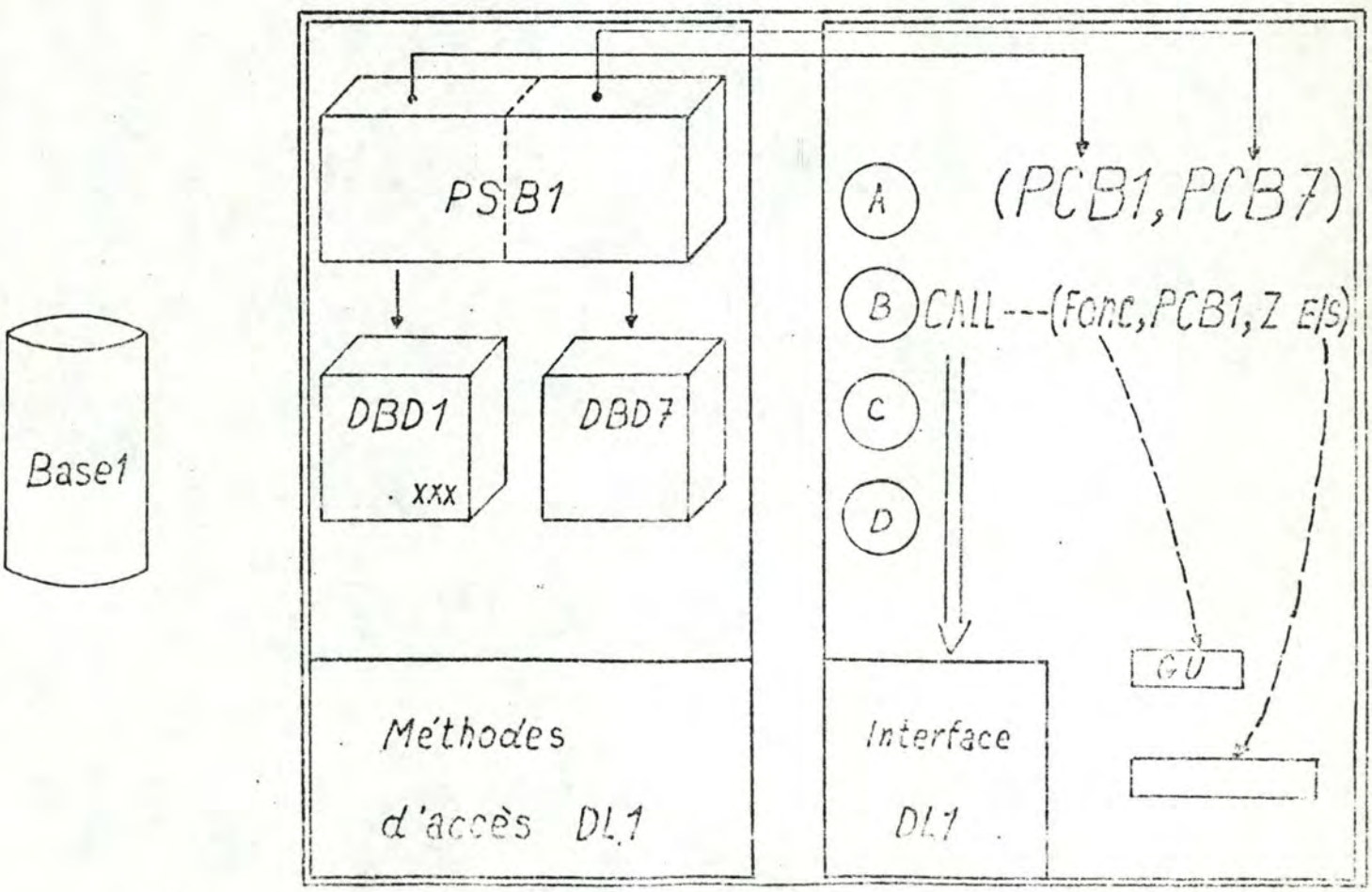


Fig. 8.8 : Appel de DL1 par un ordre CALL

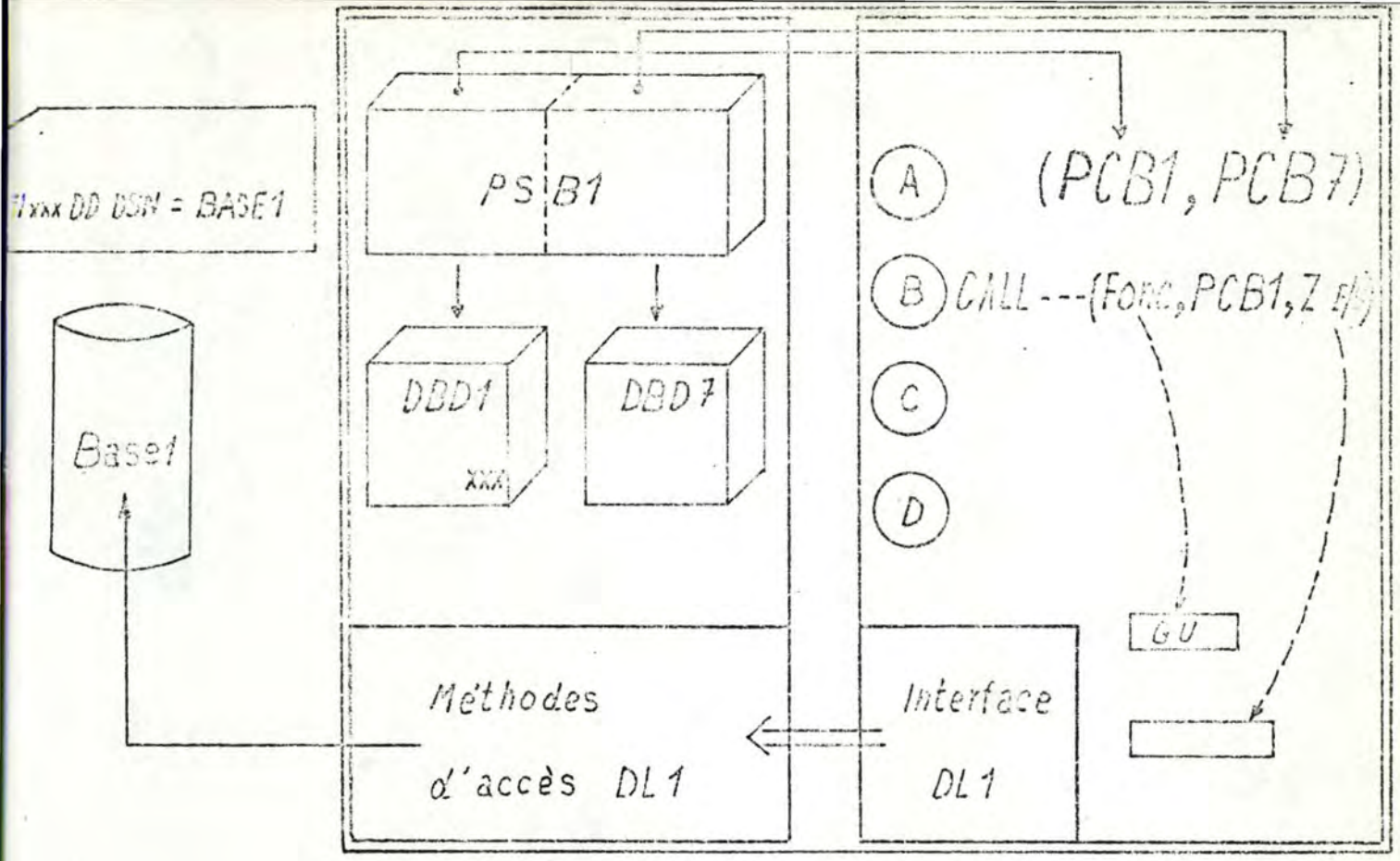


Fig. 8.9: DL1 exécute la fonction demandée et redonne le contrôle au P.A.

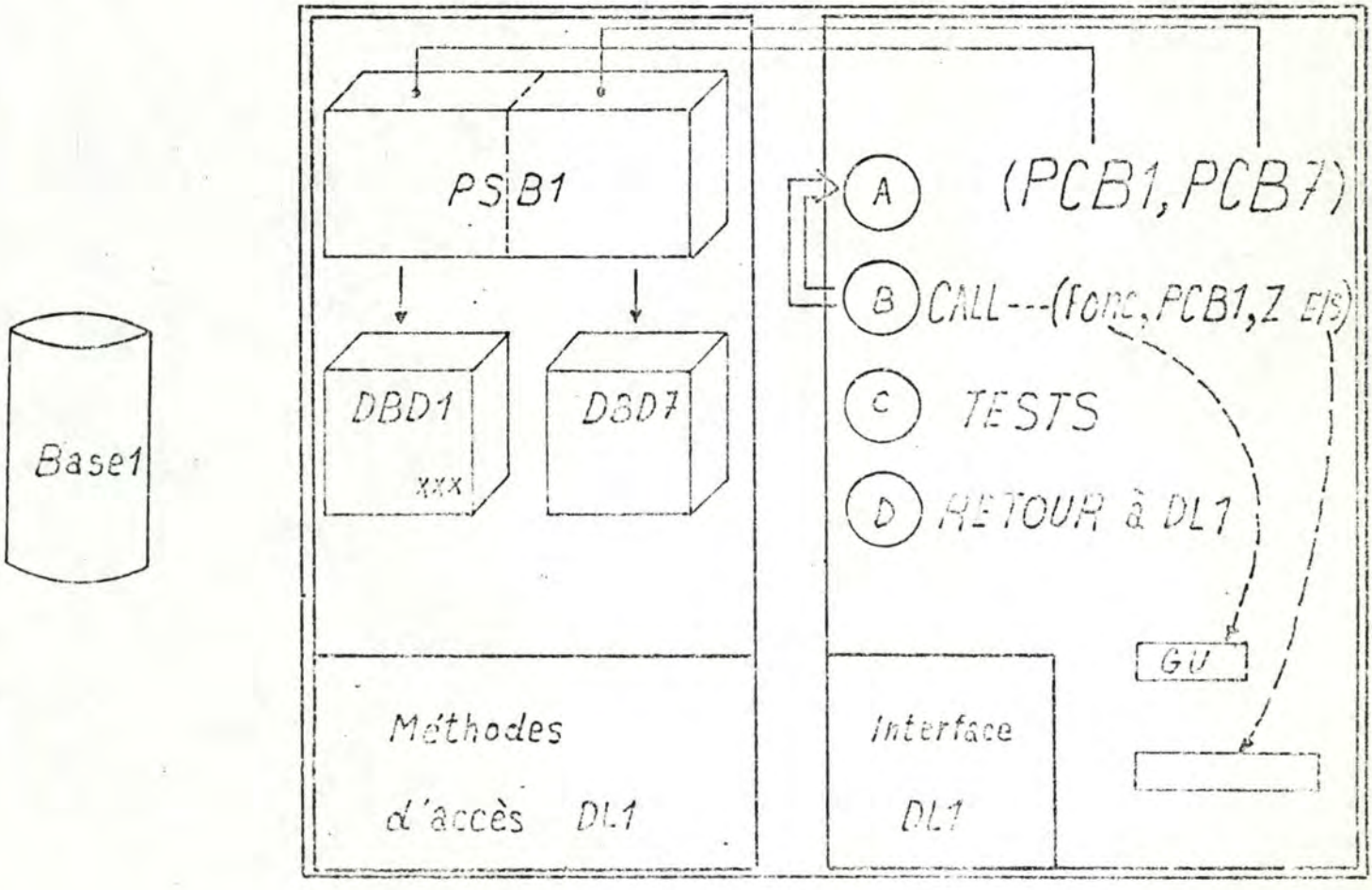


Fig. 8.10: Le programme teste les résultats de son appel

La figure précitée rappelle le mécanisme de la liaison DBD/BD. Le symbole xxx présent dans le DBD se retrouve dans une carte de contrôle associée avec les éléments permettant d'identifier le data-set, le volume et l'unité. Si tout se passe bien, l'occurrence est renvoyée dans la zone d'entrée/sortie. Sinon, d'autres codes peuvent être renvoyés: erreur d'entrée/sortie, segment non-trouvé, fin de base de données, etc. Dans tous les cas, le programme d'application reprend en séquence après l'instruction CALL.

7. (Fig. 8.10) Le programme doit s'interroger maintenant sur la manière dont son ordre CALL a pu s'exécuter. Avant tout, il analyse le code retour. Celui-ci se trouve dans le PCB qui a également une information très importante qu'on appelle clé concaténée. Cette clé n'a pas toujours la même longueur. Aussi, une autre zone du PCB a été prévue pour l'indiquer. Code retour, clé concaténée, longueur de cette clé et autres informations se trouvent à des déplacements constants par rapport au début des PCB's, dont le programme connaît les adresses. Il est donc facile d'accéder à ses précieuses informations.

8. Le programme peut exécuter autant d'ordre CALL que cela est nécessaire. Il peut par exemple boucler sur un ordre GN jusqu'à l'obtention du code retour 'FIN DE BD'. Pour s'achever, ce programme doit redonner le contrôle à DL1 dont il n'est qu'un sous-programme. DL1 donne le contrôle à son tour au système d'exploitation après avoir exécuté un certain nombre d'opérations: par exemple, la fermeture des data-sets contenant les BD's traitées par le programme d'application qui vient de s'achever.

8.8. L'index secondaire

Nous allons présenter la notion d'index secondaire en nous basant fortement sur l'exemple représenté par la figure 8.11.

La notion d'index secondaire fournit une flexibilité supplémentaire d'accès. Chaque index secondaire représente un chemin d'accès à l'enregistrement, différent de celui via la clé du segment racine. Le chemin d'accès additionnel peut résulter d'un retrait plus rapide de données. Par exemple, les segments 'PIECE' et 'COMMANDE' peuvent être retirés sur base du numéro de commande dans le segment 'COMMANDE', si un index est défini pour ce champ.

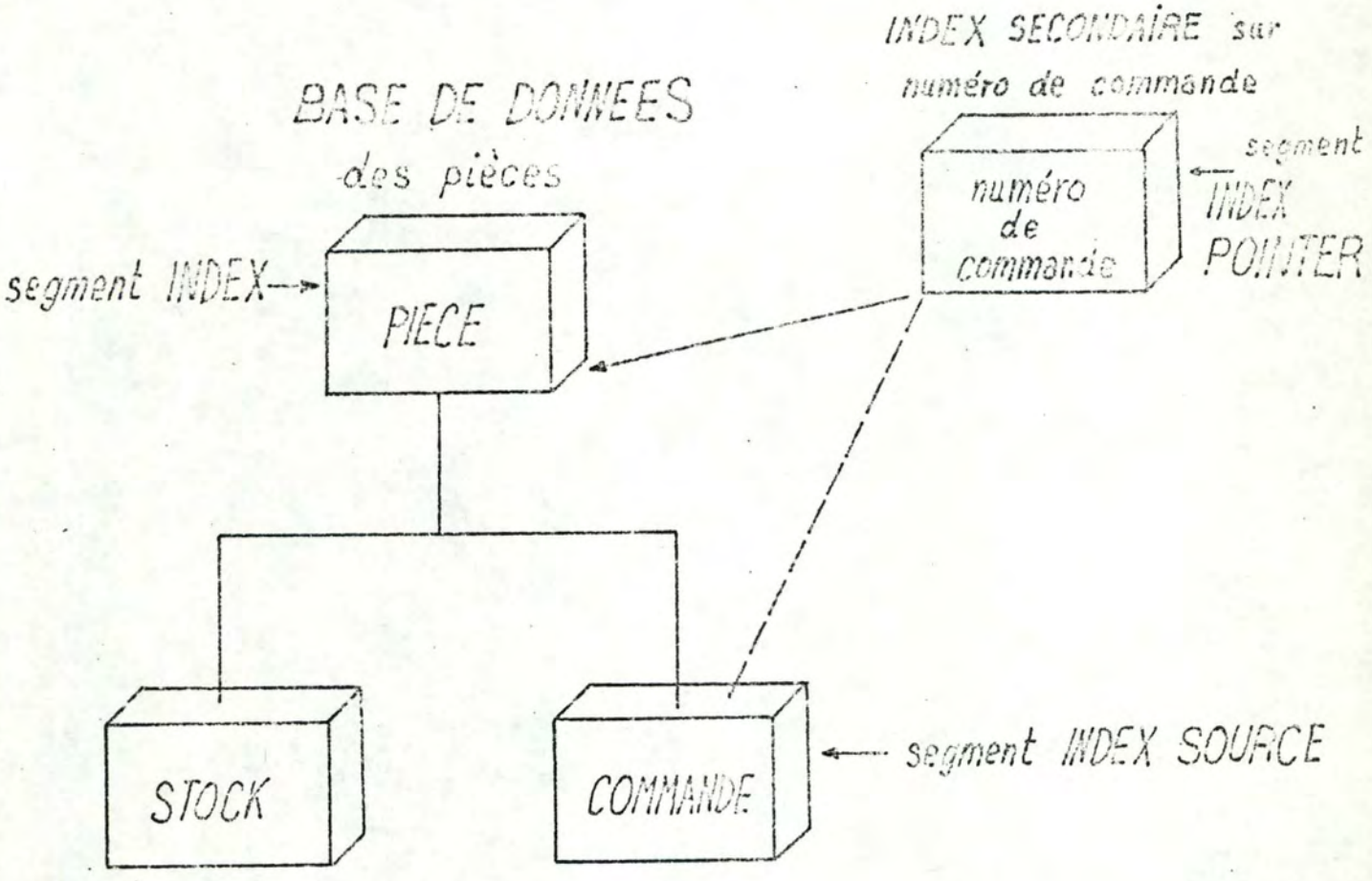


Fig. 8.11 : L'index secondaire (tiré de [IBM2])

Une fois qu'un index est défini, DL/I maintient automatiquement cet index si la donnée sur laquelle l'index est relié change.

Les segments impliqués dans un index secondaire sont mis en image par l'exemple.

Le segment INDEX SOURCE contient le(s) champ(s) source sur lesquels l'index est construit. Par exemple, numéro de commande.

Le segment INDEX POINTER est le segment dans la base de données index secondaire qui pointe le segment INDEX TARGET. Les segments INDEX POINTER sont ordonnés et accédés sur base du contenu du (des) champ(s) du segment INDEX SOURCE. Par exemple, le numéro de commande. Ceci est la séquence secondaire de parcours de la base de données indexée des commandes. S'il y a en général un segment INDEX POINTER, pour chaque segment INDEX SOURCE, de multiples segments INDEX POINTER peuvent référencer le même segment INDEX TARGET.

Le segment INDEX TARGET est le segment qui devient directement accessible via l'index secondaire. Il est dans le même enregistrement hiérarchique que le segment INDEX SOURCE et est pointé par le segment INDEX POINTER dans la base de données indexée. Souvent, mais pas nécessairement, c'est le segment racine.

Les segments INDEX SOURCE et INDEX TARGET peuvent être les mêmes, ou le segment INDEX SOURCE peut être un dépendant du segment INDEX TARGET.

Dans le cas de la présence d'index secondaire(s), il faut définir un PCB pour la base de données "normale" et un PCB pour chaque base de données index secondaire.

Le premier PCB sert au traitement en séquence primaire, tandis que chacun des seconds sert à chaque traitement en séquence secondaire.

CHAPITRE 9: Le DB/DC Data Dictionary System

9.1. Le choix d'une implémentation

Pour effectuer l'analyse des définitions de rapports, nous devons avoir accès à certains renseignements sur les Bases de Données de la société. En effet, nous devons pouvoir vérifier l'existence d'un tel type de segment d'une base de données (BD) mentionnée ou d'un tel type de champ pour un tel type de segment d'une telle BD. Il nous est nécessaire également de connaître la structure de chaque BD, pour établir le plus performant chemin d'accès à un segment. C'est pourquoi nous devons avoir accès à un ensemble de méta-données.

Il nous reste à choisir entre l'implémentation d'un répertoire propre à notre application ou l'utilisation du dictionnaire de données d'IBM, le DADIC, dont une version est implémentée à l'ARBED. Un effort étant effectué au niveau de l'ARBED pour généraliser l'emploi de celui-ci, nous avons opté pour l'utilisation du DADIC, nous épargnant du même coup tout le travail d'étude et d'implémentation d'une telle ressource.

C'est pourquoi, nous allons profiter de ce chapitre pour présenter une description générale du DADIC. Celle-ci en donne une définition, poursuit par un aperçu des éléments et des primitives qu'il propose, termine par les ponts vers d'autres sources de renseignements et les rapports qu'il peut produire.

9.2. Description générale

L'IBM DB/DC Data Dictionary est un produit qui correspond au modèle entité association. Le dictionnaire décrit des entités qui peuvent avoir des attributs et qui peuvent participer à des relations binaires "many-to-many". Ces relations peuvent posséder également des attributs. Dans la seconde version du dictionnaire, les types d'entités et les types de relations furent fixés dans le design du produit et reflétèrent les types d'entités connus dans une installation: bases de données, segments, champs, programmes, PCB's, PSB's, ... Par la suite,

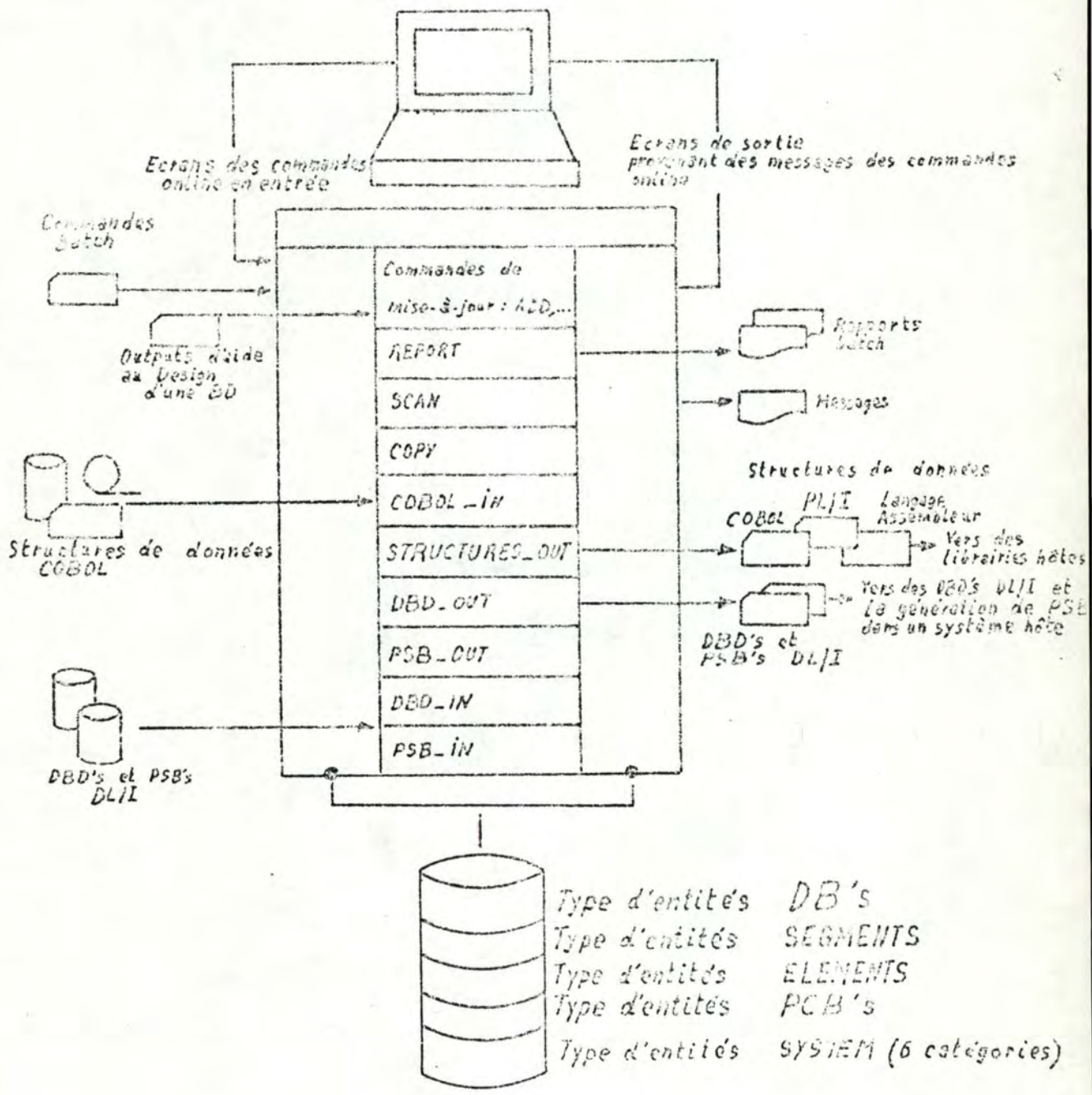


Fig. 3.1 : Le DB/DC Data Dictionary (traduit de [LEF])

le dictionnaire a fourni une facilité d'extension, qui autorise l'utilisateur à définir arbitrairement des types d'entité et de relation. Ainsi, le DD a la puissance de modélisation d'un SGBD généralisé.

9.3. La déclaration des entités et de leurs attributs

Le DB/DC Data Dictionary propose des facilités pour exprimer l'existence dans la base de données des types d'entités suivants:

- des entités de données: que nous pouvons subdiviser ainsi

  * les éléments qui peuvent être
    - un item (champ): défini comme la structure de données élémentaire. Les autres structures sont composées à partir de celui-ci.
    - un groupe: défini comme un ensemble d'items ou d'autres groupes, référencés comme des sous-groupes.

  * les segments ou records qui se définissent comme des groupes qui ont les caractéristiques spéciales de participer à un fichier ou à une base de données comme une entité constituante.

  * les bases de données ou fichiers.

- des entités de processus:

  * les Program Communication Block,

  * les Program Specification Block,

  * les transactions, définies comme le traitement qu'un programme exécute entre deux point de synchronisation.

  * les modules définis comme des collections de codes exécutables et qui sont appelés par un ou plusieurs programmes et qui peuvent appeler un ou plusieurs autres modules.

* les programmes définis en termes des attributs qui lui sont associés comme un module,

* les jobs,

* le système défini comme une collection de programmes à laquelle peut être associée une fonction majeure de l'organisation.

Le système dictionnaire de données DADIC, ne reconnaît pas l'existence d'entités utilisateurs.

Avant de présenter les attributs qui peuvent être assignés aux types d'entités et aux relations qui peuvent exister entre eux, nous allons nous arrêter sur la convention de nom pour les entités.

9.3.1. La convention de nom

Le nom d'une entité dans le DADIC est composée de quatre parties: le nom utilisateur de l'entité et trois qualificatifs du nom de l'entité. Ainsi le nom dans le dictionnaire est de la forme suivante (qualificatif de statut, code du type d'entité; nom utilisateur, occurrence).

Le nom utilisateur consiste en un maximum de 31 caractères alphanumériques. Le qualificatif de statut est un code sur un caractère qui désigne le statut d'entité de test ou de production. Le code du type de l'entité est un code sur un caractère qui a différentes significations pour des types d'entité différents. Pour le type d'entité item ou record, le code du type d'entité désigne le langage de programmation associé et identifie l'entité comme appartenant à une base de données DL/I ou à un fichier non-DL/I. Pour une entité base de données, le code identifie le type de la base ou du fichier de données. Pour les autres types d'entités, le code est simplement une lettre identifiant le type d'entité (ex: T pour transaction, ...).

L'occurrence est un nombre qui désigne l'entité comme étant une des multiples occurrences d'un nom d'entité qui autrement serait identique. Elle n'implique pas que les attributs de telles entités soient similaires.

Une entité de type élément de segment, base de données, ou PCB peut avoir non seulement un nom primaire, mais également un nom secondaire (ou déclaré alias). Ce dernier possède le même statut que le nom primaire, mais les trois autres parties peuvent être différentes. L'entité dans le dictionnaire de données peut aussi bien être spécifiée dans une commande par le nom primaire que le nom secondaire.

9.3.2. Les entités du dictionnaire et leurs attributs

Quel que soit le type d'entité du DB/DC Data Dictionary, les attributs suivants peuvent être déclarés comme partie de la définition de l'entité dans le dictionnaire:

- une description,
- des segments de données utilisateurs.

D'autres attributs sont dépendants du type de l'entité.

La description

Pour chaque entité dans le dictionnaire de données, une description en format libre peut être entrée. Celle-ci peut contenir un maximum de 999 lignes et chacune d'elles se compose d'un maximum de 72 caractères.

Les segments des données utilisateur

A chaque entité dans le DADIC, on peut associer jusqu'à cinq attributs dont le contenu et leur signification sont laissés libre à l'utilisateur. Chaque attribut peut contenir un maximum de 999 lignes de 80 caractères et l'utilisateur y introduit tous les renseignements qu'il désire.

Nous allons présenter succinctement les différents types d'entités qui peuvent rentrer dans le DADIC, ainsi que les relations entre entités de types variés. La liste des attributs que nous présentons pour chaque type d'entité n'est pas nécessairement complète et nous renvoyons le lecteur intéressé à la documentation du constructeur.

L'élément

Un élément dans le DADIC peut aussi bien être un item ou un groupe. La spécification du groupe est achevée en définissant une relation entre deux éléments et en spécifiant l'attribut 'CONTAINS' dans la relation.
En plus des attributs globaux que sont la description et les segments de données utilisateurs, d'autres peuvent être assignés à l'élément comme une spécification de longueur en bytes, le code du type de l'élément ou des attributs spécifiques aux descriptions de données PL/I.

Le segment

Le type d'entité segment est utilisé pour des segments DL/I ou des records non-DL/I. La description PL/I d'un segment est définie de la même manière que pour les éléments, mais d'autres attributs peuvent également être spécifiés.

La base de données

Le type d'entité 'base de données' est utilisé pour des bases de données DL/I ou des fichiers non-DL/I. Les attributs utilisés dépendent du type de la base de données (physique ou

logique) à décrire. Les attributs structurels d'une BD sont entrés dans le DADIC comme attributs de relations BD-segment.

Les attributs d'une BD spécifient notamment le mode d'accès, ...

Le PCB

Le PCB de DL/I décrit la collection de données à laquelle un programme accède. Le nom dans le dictionnaire d'un PCB doit inclure le nom du PSB auquel le PCB appartient. Le numéro d'occurrence spécifie la position séquentielle du PCB dans le PSB. La définition du PCB dans le DADIC spécifie le type du PCB (BD, GSAM, ou teleprocessing). Les attributs du PCB varient en fonction du type de celui-ci.

Le PSB

Les attributs spécifient notamment le langage de programmation, la plus large région I/O utilisé par un programme d'application, ...

La transaction

Les attributs indiquent le type de transaction, le type de terminal logique.

Le programme

Le programme possède comme attribut le langage dans lequel le programme est écrit, la taille du programme, ...

Le module, le job

Ces deux derniers types d'entité n'ont comme attributs que la description et les segments de données utilisateurs.

Les attributs des relations

Il existe quatre relations qui ont des attributs:

- la relation élément / élément,
- la relation segment / élément,
- la relation base de données / segment,
- la relation PCB / segment.

Les attributs de la relation base de données / segment tombent dans cinq catégories générales:

* des attributs structurels,
* des attributs physiques du segment,
* des attributs du segment enfant logique,
* des attributs du segment parent logique,
* des attributs du segment source.

Les attributs de la relation PCB / Segment consistent en des caractéristiques générales (parent et options de traitement) et la définition d'index utilisés dans les SSA's.

Il existe trois types de relation élément à élément
(de type CONTAINS: pour signifier qu'un élément en contient d'autres,
de type DEPENDS: pour signifier qu'un élément dépend d'un autre,
de type REDEFINES: pour redéfinir un élément).

L'attribut spécifie la position relative de l'un par rapport à l'autre.

Dans le cas de la relation segment à élément, les attributs indiquent la position de l'élément dans le segment et le fait qu'il soit ou non un champ de séquence connu d'IMS.

9.4. Les primitives

Nous allons reprendre ici l'objectif de chacune de ces opérations et ne présentons en aucun cas la syntaxe et la sémantique générale de chacune d'elle. Le lecteur intéressé pourra trouver tous les détails les afférant dans le DB/DC Data Dictionary User's Guide.

Les commandes valables pour l'introduction et la maintenance du dictionnaire sont classées dans trois groupes fonctionnels:

1. des commandes qui opèrent principalement sur des entités du dictionnaire et leurs attributs,
2. des commandes qui opèrent principalement sur les relations entre les entités.
3. des commandes qui opèrent sur les entités du dictionnaire qui forment des structures hiérarchiques.

9.4.1. Les commandes sur les entités simples

9.4.1.1. ADD

Cette commande est utilisée pour les propos suivants:

- entrer une nouvelle entité dans le dictionnaire de données (DD),
- entrer une nouvelle entité plus ses attributs dans le DD,
- entrer de nouveaux attributs pour une entité existante dans le DD.

9.4.1.2. CHANGE_IN

La commande change les attributs existants d'une entité.

9.4.1.3. CHANGE_NAME

Cette commande est utilisée pour changer un nom d'entité existant en un nouveau nom d'entité. Si le nom existant est un nom primaire, tous les attributs des entités existantes sont transférés à l'entité avec le nouveau nom.

9.4.1.4. DELETE

La commande détruit une entité spécifiée du dictionnaire de données.

9.4.1.5. DELETE_DATA

La commande détruit les attributs d'une entité existante autres que les attributs de relation.

9.4.2. Les commandes sur les relations

9.4.2.1. ADD_RELATIONSHIP

Cette commande a des fonctions multiples comme:

1. créer des noms secondaires pour des BD's, des segments, des champs et des PCB's,

2. établir une relation entre deux entités,

3. ajouter des attributs à une relation existante,

4. établir une relation entre attributs,

5. dans l'exécution de (1) à (4), entrer un ou les deux noms d'entité dans le dictionnaire de données.

9.4.2.2. CHANGE_RELATIONSHIP_DATA

Cette commande change les attributs existants d'une relation.

9.4.2.3. DELETE_RELATIONSHIP

Cette commande détruit une relation établie précédemment entre deux entités et tous les attributs de la relation, mais n'affecte en rien les définitions d'entité.

9.4.2.4. DELETE_RELATIONSHIP_DATA

La commande est utilisée pour détruire des attributs d'une relation pour des relations de segment à BD ou segment à PCB.

9.4.2.5. RELOCATE

La primitive autorise le changement d'une structure hiérarchique d'une BD ou d'un PCB en modifiant la séquence de leurs segments. Des segments subordonnés peuvent ainsi être spécifiés pour être mouvementés tel un groupe.

9.4.3. Les commandes sur des entités qui forment des structures hiérarchiques

9.4.3.1. DELETE_STRUCTURE

La commande est utilisée pour détruire une structure hiérarchique entière commençant à l'entité spécifiée et continuant au niveau inférieur spécifié.

Si une entité est utilisée dans une autre structure, ses relations et les attributs de ses relations dans la structure spécifiée sont détruites,

mais l'entité reste dans le dictionnaire. Si une entité est utilisée uniquement dans la structure spécifiée, elle sera détruite du dictionnaire ainsi que tous ses noms secondaires. Les exceptions à ceci sont:

- un PSB n'est pas détruit s'il est en relation avec plus d'une entité de niveau supérieur dans la hiérarchie.

- une BD n'est pas détruite si elle est en relation avec plus d'une entité de type d'entité PCB ou programme.

9.4.3.2. COPY

La commande copie tous les attributs, y compris ceux des relations si cela est demandé, pour une entité spécifiée. Elle enregistre ces attributs sous une nouvelle entité dans le dictionnaire.

9.5. Les rapports du Data Dictionary

Les facilités de rapports peuvent être invoquées par les commandes REPORT et SCAN. Des options peuvent être prises pour chacune de ces commandes afin de produire un certain nombre de rapports sur les contenus du dictionnaire. Ces rapports peuvent être présentés sur une imprimante ou à un écran de terminal.

9.5.1. La commande REPORT

Cette commande est utilisée pour produire les types suivants de rapports:

- la définition entière ou des détails sélectionnés pour une entité donnée,

- tous les noms d'entité pour un type d'entité donné,
- des équivalents imprimés des rapports sélectionnés de manière interactive à l'écran,
- des glossaires de toutes les entités d'un type donné, avec des définitions résumées.

9.5.2. La commande SCAN

Cette commande est utilisée pour analyser des noms d'entités, descriptions, segments utilisateurs ou relations d'entités avec un type d'entité spécifié pour trouver une ou deux chaînes de caractères spécifiées. Différentes options sur le contenu du rapport sont proposées.

Ainsi le rapport peut être constitué

- soit d'une liste des noms primaire et secondaire, de toutes les entités qui satisfont le critère de recherche,
- soit des noms des entités, du texte de l'attribut de chaque entité et des noms secondaires de l'entité.
- soit d'un rapport complet sur chaque entité.

9.6. L'interface avec l'extérieur

Il est possible d'écrire des programmes dans un langage conventionnel (PL/I, COBOL, Assembleur) qui accèdent au DADIC par l'intermédiaire d'une interface. Cette possibilité est offerte principalement pour permettre la création de rapports autres que ceux définis par les commandes du DADIC, notamment pour les types d'entité définis par l'utilisateur.

Les programmes peuvent être exécutés, soit on-line à partir d'une commande lancée à l'intérieur du DADIC, soit en batch. De plus, ils ne

peuvent accéder à d'autres bases de données que le DADIC. L'accès au DADIC ne peut se faire qu'en lecture (pas de modifications). Malgré ces limitations, nous avons choisi d'utiliser le DADIC comme Méta-base de données et c'est par cette interface que nous y accédons.

Les primitives prévues par l'interface sont l'accès à des listes d'entité d'un certain type, l'accès par la relation synonyme, l'accès par une relation, l'accès aux attributs d'une entité et d'une relation, l'accès aux attributs "TEXT" d'une entité.

9.7. Des possibilités de ponts

Le DADIC a des possibilités de pont avec l'environnement de définition de données d'IMS et des langages de programmation. Ces ponts existent pour générer des définitions ou entrer des données depuis des définitions existantes.

Dans l'environnement d'IMS, les commandes suivantes sont utilisées pour générer des définitions de données d'entités conservées dans le dictionnaire:

- DBD-OUT: le nom d'entité traité avec cette commande doit être du type d'entité base de données.

- PSB-OUT: la commande est utilisée pour générer du code source décrivant un PSB existant dans les entités du dictionnaire.

Les commandes possibles pour générer(en mode batch uniquement) des entités du dictionnaire depuis des DBD's et PSB's existants s'identifient ainsi:

- DBD-IN: cette commande est utilisée pour créer des entités de DBD dans le dictionnaire depuis la librairie adéquate.

- PSB-IN: cette commande accède à la librairie DL/I des PSB, retire les PSB's désignés et crée dans le dictionnaire de données les entités appropriées de PSB et PCB, ainsi que leurs relations propres.

  Un mot-clé optionnel peut être inclus dans la commande. Sa présence provoque la création non seulement d'une entité PSB, mais également d'une entité programme avec le même nom dans le dictionnaire, sauf le qualificatif du code du type d'entité. Cette dernière entité est liée avec le PSB et toutes les autres entités appropriées.

  Avant que cette commande puisse être exécutée, toutes les bases de données associées et les segments doivent exister dans le dictionnaire. Si ce n'est pas le cas, les entités ne sont pas générées et une liste des entités manquantes est produite.

  Dans l'environnement des langages de programmation, les facilités de pont consistent en la commande suivante:

  - STRUCTURE_OUT: qui recherche les entités de segment et les types d'éléments de type dans le dictionnaire. La commande est donc utilisée pour générer des structures de données dans les du langages de programmation.(Assembleur, Cobol ou PL/I).

9.8. Mesures de sécurité

Il n'existe aucune procédure dans le DADIC pour protéger le dictionnaire d'un accès non-autorisé. Dans le cas des facilités on-line, les procédures de sécurité des terminaux permettent uniquement de limiter l'accès à un certain ensemble de terminaux.

CHAPITRE 10: L'ANALYSEUR

10.1. La modularité de l'analyseur

La définition de rapport soumise à l'analyseur est un amalgame dérivé de trois langages (PL/I, langage de définition de zones, LDA). Chaque construction de la définition de rapport va subir des transformations au fur et à mesure que s'exécutent les différentes phases de l'analyse.

L'analyse lexicale consiste à transformer la vision qu'a l'analyseur de la définition de rapport d'une suite de caractères en une suite de symboles.

L'analyse syntaxique contrôle la syntaxe de toute construction et crée une représentation interne de chacune d'elle.

L'analyse sémantique vérifie certaines conditions sur chaque construction afin d'empêcher une infinité d'exécutions indéterminées.

Ces trois phases de l'analyse s'exécutent aussi bien sur les instructions classiques, les instructions d'extractions ou celles d'édition. Elles se déroulent en parallèle.

Cependant, au niveau des instructions d'extraction, l'analyse ne se clôture pas ainsi. L'analyse du LDA nécessite encore une transformation de ces expressions LDA en leur équivalent dans le modèle du MAG, si possible optimisé.

Dans la réalisation de l'analyseur, nous avons essayé de conserver des séparations entre l'analyse des différentes formes d'introduction d'une définition, afin de permettre facilement l'ajout ou le retrait d'éventuels langages ou composants.

10.2. L'analyse lexicale

La définition d'un rapport suivant le LEDGR est une suite de symboles, soit de base, soit choisis par l'utilisateur. Toutefois, le processeur considère le fichier standard dans lequel la définition de rapports a été introduite comme une suite de caractères et le traite comme telle. Il y a donc opposition.

La vision du processeur n'est pas satisfaisante et ne facilite pas l'analyse syntaxique. Aussi, avant de procéder à cette analyse syntaxique, nous opérons une transformation du fichier standard d'entrée qui constitue l'analyse lexicale.

L'analyse lexicale a pour rôle de reconstituer à partir de la suite de caractères du fichier standard en entrée, une suite de symboles de base. Chaque symbole de base a une représentation externe dans le fichier standard en entrée qui se définit comme une suite finie de caractères spécifiques à chaque symbole.

L'analyseur syntaxique ne peut manipuler directement les symboles de base. C'est pourquoi, nous profitons de l'analyse lexicale pour donner à chacun d'eux une représentation interne sous une codification plus concise et plus parlante.

10.3. L'analyse syntaxique

10.3.1. Introduction

Dans ce qui suit, l'utilisation du terme "construction" désigne à la fois les instructions, les expressions et les expressions de désignation.

L'analyse syntaxique consiste à vérifier que le fichier d'entrée contient bien une suite de symboles de base figurant la représentation externe d'une définition de rapport suivant les règles de construction du langage LEDGR. Au fur et à mesure de l'avancement des vérifications syntaxiques, l'analyse construit une représentation interne du programme

dans laquelle nous ne conservons que les caractéristiques du type de la construction et nous éliminons tous les détails d'un langage amical pour l'utilisateur. Cette représentation interne est fortement similaire à celle présentée par B. LECHARLIER lors de son séminaire (LE).

Si la chaîne de caractères à analyser n'est pas la représentation externe d'une définition de rapport LEDGR, l'analyseur syntaxique doit signaler cet état de fait par l'impression d'un message d'erreur le plus explicite possible. Cependant, il ne doit pas s'arrêter là et doit poursuivre l'analyse à la recherche d'autres erreurs afin d'apporter à l'utilisateur le plus de renseignements possibles sur ces erreurs commises dans l'ensemble du programme.

10.3.2. La représentation interne

Comme nous l'avons déjà dit, le fichier d'entrée n'est pas sous une forme propice aux traitements ultérieurs et l'analyseur y remédie en construisant progressivement la représentation interne du rapport.

A chaque construction du programme, nous faisons correspondre une représentation interne. Aussi, la représentation interne du programme se définit par une structure arborescente des représentations internes des constructions qui composent le rapport. La racine de la structure est tenue par la représentation interne de la déclaration du programme en lui-même. Chaque représentation d'une construction, élémentaire ou composée, consiste en une variable dynamique, structurée. Les niveaux mineurs de chaque variable renferment les informations nécessaires ou les adresses des informations utiles à une représentation conforme du programme en arbre.
Nous n'allons pas reprendre dans ce texte toutes les déclarations de ces variables dynamiques, mais le lecteur intéressé peut les retrouver dans l'annexe propre à ce chapitre avec la signification des composants de chaque structure.

10.3.3. Les principes de l'analyse syntaxique

10.3.3.1. Le déroulement normal

Du point de vue de l'implémentation, l'analyseur syntaxique se présente comme un ensemble de fonctions PL/I capables d'effectuer l'analyse syntaxique des constructions du langage LEDGR appartenant à une catégorie syntaxique bien déterminée (expression arithmétique, instruction, déclaration, ...). Chacune de ces fonctions est appelée dans un contexte tel que la suite des symboles à traiter doive débuter par une construction de la catégorie syntaxique correspondante, si le programme analysé est correct.

10.3.3.2. Le rattrapage des erreurs

Nous allons imaginer que le programme LEDGR à analyser contienne ne fusse qu'une seule erreur. Aussi, au cours de l'analyse, il arrive tôt ou tard, un moment où son exécution produit un appel de fonction, alors que la suite de symboles à traiter ne comporte pas entièrement une construction de la catégorie syntaxique appartenant à celle attendue.

Découvrir une erreur de l'utilisateur est une chose, mais il s'avère que lorsqu'un programme contient une erreur, cette dernière n'est souvent pas unique. Aussi, c'est fournir peu de renseignements à l'utilisateur que de lui signaler uniquement la première d'entre-elles.

C'est pourquoi, en cas de détection d'une erreur, on tente de poursuivre l'analyse syntaxique, après avoir récupéré l'erreur. Cette récupération

consiste à parcourir la suite fichier d'entrée à la recherche d'une suite de symboles qui serait corrects par rapport à la définition des constructions admises par le langage, si l'utilisateur ne s'était pas trompé.

Cependant, cette récupération ressort plus d'une devinette à propos de ce que le programmeur veut faire qu'à une procédure répondant à des règles bien strictes. Aussi, nous imposons seulement les opérations suivantes en cas d'erreur:

- l'impression d'un ou plusieurs messages d'erreur ainsi que la mémorisation de la présence d'une erreur dans le programme LEDGR à analyser. Le fait qu'une erreur, ou plus, soient détectées est mémorisé au moyen d'une variable entière, locale à l'analyseur syntaxique, mais globale aux fonctions utilisées. Cette variable conserve une valeur nulle tant qu'aucune erreur n'est détectée au cours de l'analyse.

- la poursuite du traitement de la suite des symboles à traiter après avoir retrouvé une catégorie syntaxique.

- l'exécution de la fonction appelée se clôture par le renvoi d'un pointeur vers la représentation interne d'une construction (quelconque) appartenant à la catégorie syntaxique associée à la fonction.

10.4. L'analyse sémantique

Le but de cette analyse est de déceler des erreurs pouvant se produire lors d'une infinité d'exécutions.

10.4.1. Les types d'exécution

L'exécution est définie comme l'exécution d'une suite d'actions élémentaires. Elle se détermine en appliquant aveuglément les règles sémantiques de la définition du langage. Trois types d'exécutions se distinguent.

A tout instant de l'exécution, les règles du langage définissent sans ambiguïté l'action suivante à exécuter, jusqu'au moment où la dernière est accomplie. La suite des actions exécutées est finie. Elle détermine exactement le résultat fourni par le programme. L'exécution est dans ce cas dite finie.

A tout instant de l'exécution, les règles du langage définissent sans ambiguïté l'action suivante à exécuter et cette action existe toujours. Cependant la dernière n'est jamais atteinte. Le résultat de l'exécution du programme n'existe pas et l'exécution est dite infinie.

Après un temps fini, il n'est plus possible d'appliquer les règles du langage pour poursuivre l'exécution du programme, soit parce que la situation est ambïgue, soit parce que cette situation est expressément interdite par les règles du langage. Dans ce dernier cas, l'exécution est dite indéterminée.

La détection des exécutions infinies est impossible, si ce n'est dans certains cas particuliers. Aussi, nous ne nous y attardons pas. Mais l'objectif de cette analyse sémantique est d'empêcher tout un ensemble d'exécutions indéterminées de

se dérouler. Ces exécutions sont dues à ce que l'on appelle des erreurs sémantiques.

10.4.2. Les principes de l'analyse sémantique

L'analyseur peut fonctionnellement être divisé en deux parties:

- les déclarations,
- les instructions et expressions.

L'interface entre ces deux parties est une série de tables construites à partir des déclarations et qui constituent le contexte nécessaire dans lequel le programme (la définition de rapport) peut être analysé.

Ainsi, la première partie de l'analyse sémantique consiste en la création des tables qui seront consultées tout au long de l'analyse des instructions et expressions.

10.4.3. La structure des tables

La connaissance de la structure des tables est d'une importance fondamentale. Aussi, nous allons les évoquer.

10.4.3.1. La table des identificateurs

La table clef est la table "TIDENT". Chaque déclaration d'un identificateur, que cela soit un identificateur d'une variable ou d'une procédure, y provoque une entrée. Chaque élément du tableau se subdivise en six informations dont le sens de certaines dépend du type de l'entrée.

La première de ces informations, "IDENT", reprend le nom d'identification de l'entrée.

La seconde, "LIEN", lie ensemble toutes les entrées d'identificateurs locaux à la même procédure et ou de même niveau d'imbrication dans les variables structurées. Elle consiste en un entier qui représente l'indice dans ce même tableau de la première entrée suivante, adéquate.

Le champ "OBJET" dénote le type de l'entrée par une codification sur trois bits:

- une variable simple ('000' B),
- un tableau ('001' B),
- un record ('010' B),
- une procédure ('011' B),
- une fonction ('100' B),
- un type d'article ('101' B).

Le champ "TYPE" prend comme valeur celle contenue dans le type de la représentation interne, s'il existe; prend une valeur quelconque dans le cas contraire.

La cinquième information présentée sous le sigle "REF" a une signification qui varie au gré de chaque entrée dans le tableau. Seule, une entrée de déclaration de variable simple n'attribue aucune signification à ce champ. La déclaration d'un tableau y renferme la référence vers une entrée dans la table spécifique aux tableaux. Le record y laisse l'indice dans cette même table vers le premier de ses composants. La procédure et la fonction se servent de ce champ pour retenir l'indice vers la déclaration du premier de leurs paramètres dans la table réservée à cet effet. Le

type d'article s'en sert pour conserver un indice dans la table des segments.

Le dernier renseignement ("NIVEAU") donné par cette table est le niveau d'imbrication atteint par la procédure, fonction ou variable dans la structure des déclarations.

10.4.3.2. La table des tableaux

La table des tableaux ("TARRAY") spécifie pour chaque structure de tableaux ses bornes d'indice. Chaque entrée délimite la variation d'un indice en indiquant ses limites minimale et maximale dans les champs correspondants "BORNINF" et "BORNSUP". Pour les tableaux de dimension supérieure à 1, le champ "EREF" indique la référence dans cette même table de la mémorisation des limites de variation de la dimension suivante. Ce champ a la valeur nulle dans le cas contraire.

10.4.3.3. La table des paramètres

Cette table est similaire à celle des identificateurs. Elle s'avère nécessaire. En effet à la fin de l'analyse d'une procédure, son contexte local est écrasé dans la table des identificateurs par les entrées produites par l'analyse de la procédure suivante.

10.4.3.4. La table des articles

Celle-ci se subdivise en neuf informations. Ces différentes informations sont:

- soit nécessaire aux analyses syntaxique et sémantique:

  * un nom utilisateur,
  * une référence vers la table des items,
  * le nombre d'items pour cet article

- soit nécessaire à l'analyse des chemins

  * le nom IMS,
  * la longueur fixe de l'article,
  * le niveau de l'article dans la structure hiérarchique,
  * la référence de son parent dans cette même table,
  * la référence vers la base de données du parent logique,
  * la référence vers le segment parent logique,

10.4.3.5. La table des items

Comme la table des articles, celle-ci comprend des informations utiles

- aux analyses syntaxique et sémantique:
  * un nom utilisateur,
  * le type de cet item.

- à l'analyse des chemins:
  * le nom IMS du champ correspondant à cet item,
  * la position de début du champ dans le segment,

* une caractéristique afin de savoir si l'item est défini comme champ IMS et si celui-ci est un champ de séquence ou pas.

10.4.4. Les conditions à assurer sur chaque construction

10.4.4.1. L'expression de désignation

Si X est le nom d'une variable simple ou d'un tableau, la table "TIDENT" doit contenir une entrée de cet identificateur qui correspond au type (variable simple ou tableau).

Si X est le nom d'une variable structurée en record, la table "TIDENT" doit contenir une entrée de cet identificateur et parmi ses composants, une entrée de l'identificateur concerné.

Si X est le nom d'une zone, la table "TZONE" doit contenir une entrée de cet identificateur.

Si X est le nom d'une BD, segment, champ, une entrée doit exister de cet identificateur dans la table correspondante.

Si X est le nom de désignation d'un article ou item, une entrée doit exister dans la table des articles et/ou des items.

10.4.4.2. L'instruction d'affectation

L'instruction d'affectation se présente sous la forme dexpr = expr;

où: dexpr est une expression de désignation, et expr une expression.

Au point de vue sémantique, dexpr et expr doivent être de même type.

10.4.4.3. L'instruction conditionnelle

Quelle que soit l'une des deux formes sous laquelle elle se présente, l'analyse sémantique doit vérifier que la condition qui caractérise l'instruction est bien booléenne.

10.4.4.4. Les instructions de répétition

Dans le cas des instructions de répétition contrôlées par une condition, l'analyseur doit vérifier que cette condition est booléenne.

Dans l'autre cas, les expressions limitant l'action du compteur doivent être toutes des expressions arithmétiques.

10.4.4.5. L'appel de procédure

Si X est le nom d'une procédure, la table "TIDENT" doit contenir une entrée de cet identificateur.

Le nombre de paramètres effectifs est égal au nombre de paramètres formels.

Pour tout paramètre effectif, son expression doit être de même type que la déclaration du paramètre formel correspondant. Si le paramètre formel est un tableau, les valeurs des indices effectifs doivent être comprises entre les bornes déclarées dans la table "TARRAY" de l'indice correspondant.

10.4.4.6. L'instruction d'extraction FOR_EACH

Le type de segment sur lequel s'effectue la recherche, doit être déclaré dans la table des segments. Pour chaque clause, certaines vérifications doivent être effectuées. Ainsi, nous devons vérifier pour la clause WHERE si les champs spécifiés du segment sont déclarés dans la table des champs et effectuer l'analyse de l'expression. Pour la clause de position, nous devons la vérifier par un accès au DADIC. Si la clause d'ordre est spécifiée, la vérification porte sur l'existence ou non dans la table des champs de chacun de ceux figurant dans la liste spécifiant l'ordre. Pour la clause de groupe, la vérification porte à nouveau sur l'existence ou non dans la table des champs de chacun de ceux figurant dans la liste spécifiant le groupage.

10.4.4.7. L'instruction d'extraction FIND

Les contrôles s'avèrent être le mêmes que pour l'instruction précédente.

10.4.4.8. Les instructions de génération des zones

Pour chacune d'elle, il faut vérifier l'existence dans la table des zones de l'identificateur la spécifiant.

10.4.4.9. Les expressions

Pour l'appel de fonction, il doit exister dans la table "TIDENT" une entrée de cet identificateur.
Les vérifications à entreprendre sur les arguments sont identiques à celles pratiquées sur les paramètres d'un appel de procédure.

10.5. L'analyseur des instructions d'extraction

10.5.1. L'objectif

L'analyseur des instructions d'extraction a pour objectif de transformer un programme contenant des ordres du LDA en un équivalent s'exprimant dans les concepts du MAG. Il faut passer d'un langage de manipulation d'ensemble d'objets au travers de boucle, à un programme réalisant des accès ponctuels dans des bases de données par l'intermédiaire de primitives.

Cette transformation, effectuée sur base des informations contenues dans la méta-base de données, essaie d'être satisfaisante sur le plan des performances.

Vu les problèmes évoqués dans la section 5.3.4.2., l'analyseur est propre aux bases de données gérées par IMS.

10.5.2. L'algorithme prédicatif ou effectif

Le programme LDA peut être considéré de deux points de vue:

- soit comme l'expression du résultat désiré (algorithme prédicatif);

- soit comme l'expression à la fois de ce que l'on veut et surtout de la manière de l'obtenir (algorithme effectif).

Dans le cas d'un algorithme prédicatif, l'analyseur doit choisir un algorithme effectif équivalent. Il peut travailler de deux façons:

- considérer l'algorithme prédicatif comme étant une base de départ que l'on essaie d'optimiser. Alors, nous nous rapprochons d'un algorithme effectif.

- considérer l'algorithme prédicatif comme une pure description des objets à acquérir et à créer, de toutes pièces, un algorithme effectif qui puisse y accéder.

Le LDA que nous avons choisi d'implémenter s'apparente davantage à une description effective:

- le langage est un langage de boucle. Pour spécifier les relations entre les différents articles accédés, nous utilisons des boucles imbriquées et nous indiquons le chemin à utiliser. Chaque boucle ne spécifie qu'un seul chemin.

- les conditions de sélection d'un article se basent uniquement sur les valeurs de ses items.

Cependant, l'analyse du programme n'est pas pour autant triviale.

10.5.3. L'identification des bases de données

L'analyseur a besoin de connaître la structuration de la base de données à accéder. Ces informations sont chargées au fur et à mesure de l'analyse syntaxique du programme.

L'instruction de déclaration des bases de données doit spécifier le nom du PSB par lequel il est possible d'accéder simultanément à tous les articles. On peut utiliser un synonyme dans la mesure où il a été déclaré dans le dictionnaire de données.

A partir de ce nom, l'analyseur peut retrouver les bases de données IMS utilisables. Ensuite, il retrouve les articles, et les chemins qui les relient. Les informations sont mémorisées dans la table des articles (notamment le nom utilisateur et le nom IMS).

Il n'y a pas besoin d'avoir une table des chemins d'accès, car les renseignements se trouvent dans la table des articles.

La description d'un article comprend des informations sur les champs qui le composent. Cette description n'est chargée dans la table des items que si on fait référence au type d'article dans une boucle ou une déclaration.

10.5.4. Le format général d'une boucle

A une boucle d'accès correspond le schéma d'exécution suivant:

- une phase d'initialisation qui s'exécute une seule fois et qui comporte:
  * l'initialisation de compteurs éventuels,
  * la création de collections temporaires d'objets.

- l'accès à chaque élément de l'ensemble:
  * cet accès se fait éventuellement avec des conditions,
  * il se fait avec des actions à effectuer sur chaque élément.

- Une phase de clôture
  * qui libère d'éventuelles ressources,
  * et effectue des calculs sur les compteurs.

L'analyseur construit un descripteur de boucle permettant de mémoriser les différentes décisions prises lors de l'analyse des clauses de la boucle.

10.5.6. L'analyse des relations entre boucles

L'analyseur considère que l'algorithme qu'on lui fournit est globalement l'algorithme final. En effet les optimisations qu'il peut effectuer sont assez limitées:

- Vu qu'une boucle ne spécifie au maximum qu'un seul chemin entre des types d'articles, il n'y a pas de choix sur le chemin à effectuer.

- Un cas classique d'optimisation consiste à intervertir l'ordre des boucles afin de respecter la structure des chemins dans la base de données. Or dans le corps d'une boucle peuvent se trouver des instructions autre que celles d'accès. Si nous modifions l'ordre d'exécution des boucles, nous changeons aussi l'ordre d'exécution de toutes les instructions.

Exemple:

Supposons qu'il existe des clients et des commandes reliés par une relation. Pour obtenir les dates des commandes des clients, nous pouvons écrire deux programmes, suivant que nous accédons d'abord au client ou aux commandes:

```
1)  FOR_EACH CL = CLIENT
      DO
         /*   */
         FOR_EACH CO = COMMANDE FROM CL
              PRINT CO.DATE;
    END;

2)  FOR_EACH CO = COMMANDE
      DO
         /*   */
         FOR_EACH CL = CLIENT FROM CO
              PRINT CO.DATE;
    END;
```

Supposons que, dans une base de données IMS, les articles commandes soient stockés en dessous du client correspondant. Le second programme est moins efficace puisque nous accédons d'abord à toutes les commandes, et donc à leur client, avant d'accéder de nouveau au client.

Autant que possible, l'analyseur doit être capable de reformer l'algorithme 1 à partir de l'algorithme 2. Cependant si nous rajoutons une instruction entre les deux FOR_EACH imbriqués (à la place du commentaire dans l'exemple), nous ne pouvons exécuter la transformation puisque l'instruction serait effectuée un nombre différent de fois dans les deux algorithmes.

En fait, le problème vient du fait que derrière les boucles se cachent deux concepts:

- l'accès aux articles, et
- la spécification du chemin à suivre pour y accéder.

Pour pouvoir faire une optimisation globale, il aurait fallu que le langage fasse une plus grande distinction entre les deux concepts (comme par exemple la forme complète du LDA défini à l'Institut). Notre analyseur ne fait lui que des optimisations par boucle d'accès.

10.5.7. L'analyse de la clause WHERE

La clause de sélection est une expression booléenne formée d'un ensemble de conditions. L'expression n'est pas directement exprimable vers l'interface:

- soit elle comporte des conditions sur des items qui ne sont pas définies à IMS,
- soit la structure de l'expression n'est pas acceptable.

L'analyseur doit la scinder en deux expressions E1 et E2 telles que

- le produit de E1 et de E2 donne l'expression de départ.
- E1 respecte le structure d'une expression fournie à l'interface et est uniquement constituée de conditions sur des champs définis à IMS.
- E2 reprend le minimum de l'expression de départ et reprend les conditions à vérifier explicitement.

Ces deux expressions vont chacune être attachées au descripteur de boucle à la place de l'expression originale.

Il y a des cas particuliers:

1. Si nous spécifions une expression comportant des conditions sur un item qui est un champ IMS, source d'un index secondaire et placée dans une boucle sans relation, alors nous pouvons envisager d'utiliser cet index secondaire.

2. Si la clause se trouve dans une boucle d'accès par un lien logique, l'expression de condition ne peut pas être en partie adressée à IMS (ceci est une limitation de l'interface).

10.5.8. L'analyse de la clause d'ordre

La clause d'ordre peut spécifier un accès dans un ordre différent de celui dans lequel se trouvent les articles. Si tel est le cas, il faut accéder à l'ensemble des articles spécifiés par les autres clauses, les placer dans une liste et trier cette liste sur l'ordre demandé. Ensuite la boucle d'accès véritable est effectuée sur les éléments de la liste.

Dans la partie 'initialisation' du descripteur de la boucle, nous plaçons l'ordre de définition d'une liste. A cette liste

sont attachés un nouveau descripteur de boucle reprenant toutes les clauses de la boucle originale et un descripteur de l'ordre à donner à la liste. La descripteur de la boucle principale spécifie qu'il s'agit d'un accès sur une liste.

Détection de la nécessité d'un tri:

- Dans IMS, l'ordre des segments est celui du champ de séquence s'il existe. Cet ordre a pour référentiel le type de chemin dans lequel se trouve le type de segment:

  * toute la base de données,

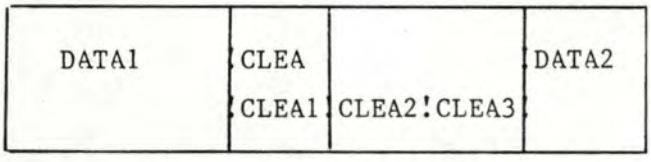
  * ou bien uniquement le chemin inter-article suivant que le segment est un segment racine ou non.

  Pour les index secondaires, il faut s'assurer que ceux-ci reprennent toutes les occurrences du segment (pas de 'sparse indexing').

- Il vaut vérifier si les champs sur lesquels l'utilisateur demande un ordre, forme une zone qui débute la zone de séquence IMS et ne laisse pas de vide.

Exemple:

Soit CLEA, le champ IMS de séquence pour ce segment, CLEA1, CLEA2, CLEA3, DATA1, DATA2, des champs utilisateurs



Tout ordre qui n'est pas (CLEA) ou (CLEA1) ou (CLEA1, CLEA2) ou (CLEA1, CLEA2, CLEA3) demande le tri des segments.

10.5.8. L'analyse clause GROUPED

La clause de groupe définit des sous-ensembles parmi les articles qui sont accédés dans la boucle. Ces sous-ensembles peuvent eux-mêmes faire l'objet d'une boucle imbriquée d'accès qui en extrait:

- soit des éléments,
- soit de nouveaux sous-ensembles.

Pour l'analyse de cette clause, nous avons besoin de parcourir les instructions du corps de la boucle afin de déterminer si elle contient des boucles d'accès sur la collection.

L'ensemble des clauses de groupe des boucles imbriquées définit un ordre sur les articles. Comme pour la clause 'ORDER', il faut vérifier si l'ordre est celui des articles dans la base de données. Dans le cas contraire, nous créons une liste triée.

S'il existe une clause 'HAVING', celle-ci nécessite un premier passage en revue des éléments des sous-ensembles afin de tester si la condition est satisfaite. Si cela n'a pas déjà été fait par l'analyse de la clause d'ordre, nous créons une liste pour stocker les éléments.

Quant il existe des boucles d'accès imbriquées, il est intéressant de n'accéder aux éléments qu'à l'intérieur de la boucle la plus interne. Ceci est possible à condition que tous les éléments de chaque sous-ensemble soient accédés dans la boucle imbriquée. Il ne faut pas de clause de sélection dans les boucles imbriquées, ni d'exécution sélective (par exemple par l'intermédiaire d'une instruction IF).

10.5.9. L'analyse des quantificateurs

Les quantificateurs d'une boucle LDA ne sont identiques à ceux du MAG que s'il n'y a pas de sélection effectuée en dehors du MAG. Dans le cas contraire, la quantification passée au MAG spécifie l'intégralité des articles (1-*), et

spécifie l'intégralité des articles (1-*), et la limitation de l'accès aux articles d'un certain rang est gérée par le test d'un compteur.

10.5.10. L'analyse d'expressions statistiques

Lorsque nous rencontrons une expression statistique, il faut signaler à la boucle qui définit son contexte d'exécution qu'elle doit:

- mettre à zéro des compteurs dans la partie 'initialisation' de sa boucle,

- éventuellement, effectuer un calcul dans la partie 'clôture' (dans le cas d'une fonction de moyenne).

Il faut signaler à la boucle qui effectue l'accès qu'elle doit mettre à jour le compteur en récupérant la valeur d'un item de l'article.

Pour récupérer la valeur de l'expression statistique, il suffit d'accéder à la valeur du compteur comme à une variable PLI.

Si on rencontre plusieurs fois la même expression statistique, il faut faire attention à ne générer qu'une seule variable.

CHAPITRE 11: La génération du code PL/I

11.1. Introduction

Une fois que le programme a été analysé, il s'agit d'en regénérer un équivalent PLI comportant des déclarations et des instructions conformes aux règles de ce langage. Comme la génération n'est globalement que l'inverse de l'analyse syntaxique, nous n'allons reprendre que les points pour lesquels il existe des particularités, c'est-à-dire:

- les structures utilisées par l'interface MAG-IMS,
- la génération des boucles d'accès,
- les variables d'articles,
- les fonctions statistiques,
- les structures utilisées pour la gestion des zones.

11.2. Les structures utilisées par l'interface

L'interface doit connaître un minimum d'informations sur les bases de données qu'il va utiliser.

Pour lui permettre d'accéder à ces informations, trois tables sont à sa disposition:

- la table des bases de données,
- la table des types de segments,
- la table des PCB's.

Les deux premières tables sont déclarées avec des clauses 'INIT' qui les garnissent lors de la compilation avec les bonnes valeurs.

La table des PCB's est garnie à l'exécution avec les adresses des PCB's fournies par IMS.

11.3. La génération d'une boucle d'accès

Soit une boucle simple sur la variable ZZ.

```
FOR_EACH ZZ = XXX WHERE (<cond>)
     <instruction>
```

Une première phase consiste à déclarer les variables suivantes:

- ZZ_VAR comme variable d'article,
- ZZ une structure PLI permettant l'accès aux valeurs d'items,
- ZZ_SSA une zone de description des conditions,
- ZZ_STATUS un indicateur.

La génération du code PLI s'effectue suivant ce schéma:

```
ZZ_PROLOGUE:
     <instructions d'initialisation>
DO WHILE ZZ_STATUS = 0% ;
     CALL MAGIMS (ZZ_VAR, ZZ_STATUS, ZZ_SSA, ...);
     IF (<conditions non-exprimables dans MAG>) THEN
     DO;
          ...
          <instructions du corps de la boucle>
     END;
END;
ZZ_EPILOGUE:
          <instructions de clôture>
```

Ce schéma général subit des variations quand l'accès nécessite un tri ou une rupture.

11.4. La déclaration d'une variable d'article

Une variable d'article est une structure PLI contenant:

- le numéro de la physique du type d'article,
- le numéro du type d'article dans sa BD,
- la clé concaténée de l'article,
- le numéro d'exécution de la boucle,
- l'adresse de la structure PLI destinée à recevoir les valeurs d'item.

Son nom est par convention le nom de la variable suivi de '_VAR'.

11.5. La génération des conditions exprimables dans le MAG

Pour passer des conditions, nous déclarons une structure PLI semblable au format des SSA's IMS.

Par exemple pour l'expression

(A <= EXP1) and (A >= EXP2)

où A correspond au champ IMS de NOM AIDBDB01 et de type CHAR (4), nous générons la déclaration suivante:

```
DCL
     01 COND001,
          02 NOMIMS001   CHAR(8)   INIT('AIDBDB01'),
          02 COOP001     CHAR(2)   INIT('<='),
          02 VALEUR001   CHAR(4)   ,
          02 RELATI001   CHAR(1)   INIT('*');
          02 NOMIMS002   CHAR(8)   INIT('>='),
          02 VALEUR002   CHAR(4)   ;
```

Dans l'initialisation de la boucle, nous générons les instructions de garnissage des valeurs:

```
COND001.VALEUR001 = EXP1;
COND001.VALEUR002 = EXP2;
```

11.6. La génération d'une fonction statistique

Nous déclarons une variable numérique dont le nom est formé par concaténation du nom de la fonction, de la boucle qui sert de référence et de la valeur d'item.

Dans la zone d'initialisation de la boucle de référence, nous mettons à zéro la variable.

Dans la boucle d'accès, on rajoute une instruction de mise-à-jour du contenu.

Exemple:

le petit programme

```
FOR_EACH ZZ = XXX
DO
     ...
END;
PRINT SUM (ZZ.ITEM1,ZZ);
```

donne lieu à la déclaration

```
DCL  SUM_ZZ_ZZ_ITEM DEC FLOAT;
```

et aux instructions d'initialisation et de mise à jour suivantes

```
SUM_ZZ_ZZ_ITEM1 = 0;
SUM_ZZ_ZZ_ITEM1 = SUM_ZZ_ZZ_ITEM1 + ZZ.ITEM;
```

Pour récupérer la valeur de la fonction statistique, il suffit de désigner la variable. Pour l'exemple, cela donne:

```
PRINT SUM_ZZ_ZZ_ITEM1;
```

11.7. La structure utilisée pour la génération de zones

Les routines qui effectuent la gestion des zones doivent en connaître la description. Celle-ci est stockée dans un tableau déclaré avec des clauses 'INIT'. L'adresse de ce tableau est passée lors de l'initialisation des routines de gestion de zones.

Les instructions de génération de zone sont transformées en appel à des procédures.

CHAPITRE 12: La gestion du concept de zone

12.1. Introduction

Une exécution d'une définition d'un rapport n'est pas spécifique à sa définition logique. Celle-ci peut varier suivant les caractéristiques des demandes et provoquer la génération d'un nombre différent d'une zone précise. La présentation physique proporement dite du rapport est donc toute-à-fait différente d'une exécution à l'autre. Aussi, un certain nombre de procédures doivent gérer le concept de zone au moment de l'exécution de la génération du rapport proprement dite. Ces procédures doivent assurer que la suite des zones générées soit logique à la définition et que ces zone s'enchaînent sans anarchie.

Cependant, nous n'allons pas développer cette gestion jusqu'à un niveau de détail permettant une implémentation rapide. En effet, le temps nous manquant, nous avons choisi de porter nos efforts sur la description algorithmique du rapport et de laisser à une étude ultérieure l'analyse détaillée de la description logique.

Ce chapitre a donc pour unique but de présenter les grands principes que notre courte réflexion a décelé à ce propos.

12.2. L'image d'un rapport stocké

Le rapport peut être considéré comme un grand tableau à deux dimensions dont nous ne connaissons pas les dimensions effectives. Chaque zone est alors au rapport, ce que représente, en algèbre, la sous-matrice dans la matrice.

Le remplissage de ce tableau ne s'effectue pas de haut en bas de manière parfaitement 'séquentielle'. En effet, des retours en arrière (vers le haut) s'avèrent nécessaires pour compléter des lignes déjà partiellement remplies. Aussi, ce remplissage de cette matrice peut être comparé à la gestion de pages virtuelles de mémoire. Ce besoin se révèle notamment dans le cas de génération d'une zone dont la hauteur

dépasse celle d'une page de liste (ou d'écran), ou encore dans le cas d'une génération d'une zone se trouvant à droite d'une autre.

12.3. Les grands principes

Le premier principe concerne la séquence de génération des différentes zones incluses dans un rapport. Celle-ci s'établit pour des zones ensemble de haut en bas, puis de gauche à droite. Toutes les occurrences d'un même type de zone sont générées avant de commencer la génération d'un autre type de zone. Ainsi, pour l'exemple illustré par la figure 12.1., elle se présente comme étant:

1a-1b-3a-3b-5a-5b-2-4-6.

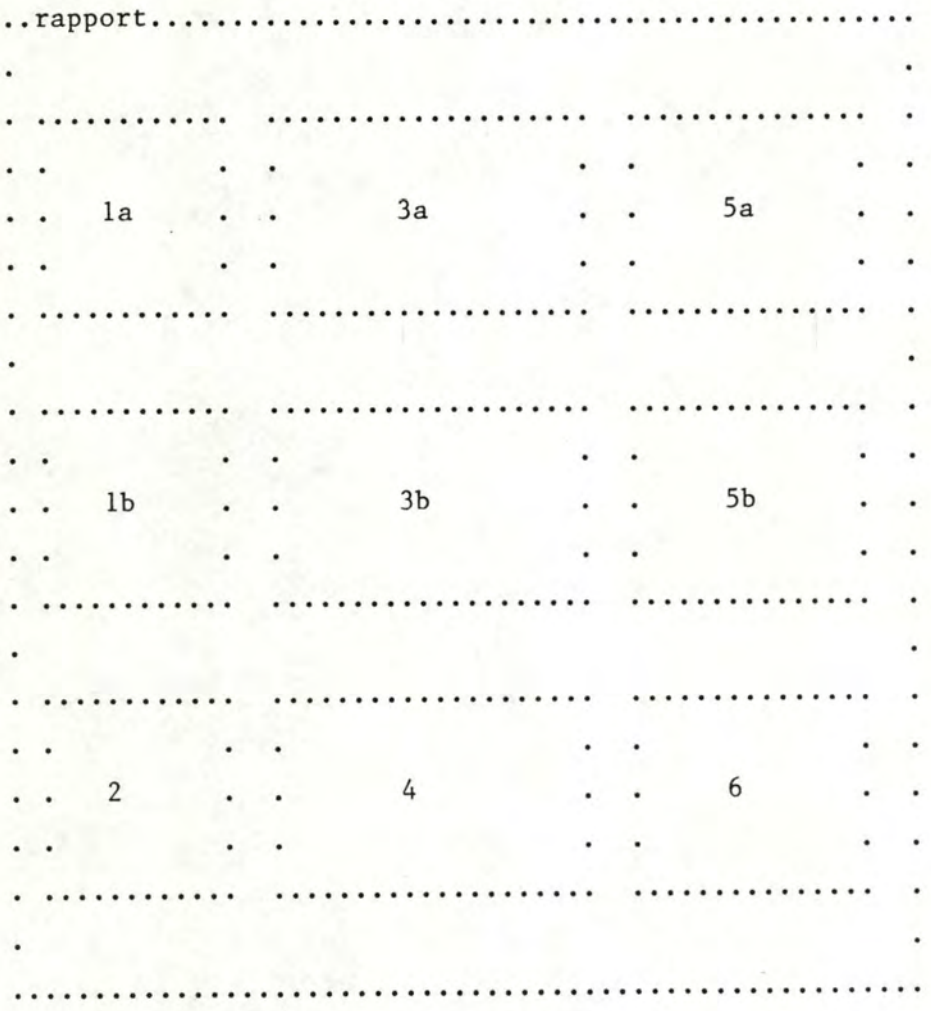


Fig. 12.1: La séquence de génération

La largeur d'une zone ne peut varier à l'exécution. Si cette restriction n'est pas imposée, il faut garder en mémoire la position de toutes les zones précédemment générées afin de la réajuster au gré d'une zone. Ce raisonnement se justifie pour un souci de parallélisme dans la génération d'un type de zone.

Le problème est moins important en ce qui concerne la hauteur variable d'une zone. Vu la séquence de génération des zones, une hauteur plus ou moins longue n'influence que la position relative des zones qui se trouvent en-dessous de celle générée. Après la génération d'un type de zone, il faut comparer la hauteur atteinte du rapport à la position de début de chaque type de zone qui se trouve en-dessous. Si cette comparaison reflète un risque d'écrasement de lignes dans la matrice de configuration du rapport, un réajustement de la position de début de la zone doit être effectué.

Nous devons tenir dans une table des informations sur chaque zone. Ces informations comprennent les renseignements présentés lors du chapitre 6 (La description logique) ainsi qu'une référence vers la description de la zone qui l'englobe. Cette référence sert, lors de la terminaison de la génération d'une zone, à réajuster la hauteur de la zone qui l'englobe directement.

Pour garantir le parallélisme entre différents types de zone il faut contenir ces premiers dans un type de zone les englobant. Dans la figure 12.2, nous illustrons ce que nous entendons par parallélisme de zones. Nous y trouvons la représentation d'un rapport qui est composé de deux occurrences des types de zones 'zone A' et 'zone B'. Chacune de leur occurrence commence à la même ligne par rapport à la position de l'occurrence du type de zone 'ss-rapport' qui l'englobe.

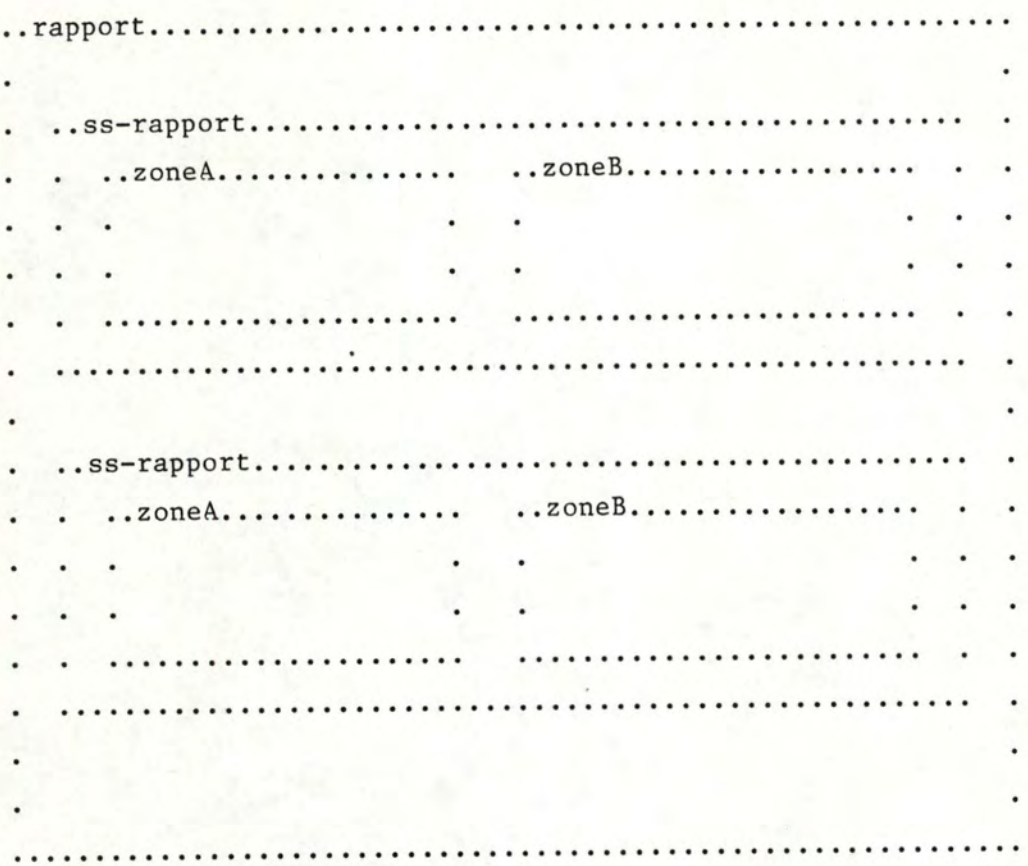


Fig. 12.2: Le parallélisme des zones

L'utilisateur peut générer sur chaque page (écran) un en-tête et/ou bas-de-page qu'il a le loisir de modifier dynamiquement au cours de l'exécution de son rapport. Aussi, des procédures doivent contrôler la génération du rapport afin d'assurer la présence de ces zones spéciales sur chaque page (écran).

Conclusions et perspectives

Arrivés au terme de ce mémoire, nous pouvons tenter une brève évaluation de ce qui a été réalisé.

L'objectif de ce travail consistait à concevoir et à implémenter un système de génération de rapports. Dans notre cas, les informations étaient tirées de bases de données IMS.

Notre ambition était de concevoir ce système de telle manière qu'il puisse accéder à des bases de données hétérogènes. Lors de notre stage, nous avons fait le choix d'utiliser le dictionnaire de données DADIC d'IBM, et nous avons refusé une implémentation personnelle d'un dictionnaire. Ce choix s'est justifié par le temps trop court qu'il nous était imparti pour effectuer cette étude par rapport à l'ampleur du sujet.

Le choix du DADIC nous a obligé a abandonné le caractère universel que nous voulions donner à notre générateur. Il ne saura accéder à des bases de données hétérogènes, mais sera orienté vers des bases de données IMS documentées dans une version du DADIC.

Du point de vue de la réalisation, une version de l'analyseur de la description algorithmique du rapport et une version du générateur sont implémentés. Cependant, le système n'aura pas de consistance tant qu'un éditeur attrayant et un analyseur de la description logique n'existeront pas. A ces deux derniers, il est indispensable d'ajouter une analyse détaillée des procédures de gestion du concept de zones. Ceci met en évidence l'énorme travail qu'il reste à effectuer avant de pouvoir évaluer pleinement l'opportunité à découper le rapport sous la forme de zones. Aussi, nous ne pouvons qu'espérer que notre étude trouvera une suite afin d'aboutir à un système réellement 'User Friendly'. Un certain nombre de lecteurs se poseront la question de savoir si le choix d'un langage procédural est judicieux pour un interface 'User Friendly'. Ce choix s'est justifié par la décision d'utiliser le Langage de Description d'Algorithmes et le Modèle d'Accès Généralisé, tous deux développés à l'Institut. Mais il est bien entendu que d'autres interfaces peuvent rendre un tel système plus attrayant.

SIGNIFICATION DES ABREVIATIONS

BD = DB	Base de Données
DADIC	DB/DC Data Dictionary
DB = BD	Base de Données
DBD	Data Base Description
DBR	Data Base Record
DD	Dictionnaire de Données
DLET	abréviation de l'ordre DL/I Delete
GHN	abréviation de l'ordre DL/I Get Hold Next Parent
GHNP	abréviation de l'ordre DL/I Get Hold Next within
GHU	abréviation de l'ordre DL/I Get Hold Unique
GN	abréviation de l'ordre DL/I Get Next
GNP	abréviation de l'ordre DL/I Get Next within Parent
GU	abréviation de l'ordre DL/I Get Unique
ISRT	abréviation de l'ordre DL/I Insert
IMS	Information Management System
LDA	Langage de Description d'Algorithmes

MAG	Modèle d'Accès Généralisé
PA	Programme d'Application
PCB	Program Communication Block
PSB	Program Specification Block
REPL	abréviation de l'ordre DL/I Replace
SGBD	Système de Gestion de Bases de Données
SSA	Segment Search Argument

B I B L I O G R A P H I E

(B) BOCK et DELVAL
Génération Automatique d'Interface d'Accès à des Bases de Données.
Institut d'Informatique, Namur, 1982 (mémoire)

(BSC) BRITISH COMPUTER SOCIETY
Data Dictionary Systems Working Party
Journal of Development
Summer 1982

(DA) DATE, C.J.
An Introduction to Database Systems.
Addison-wesley, 1981

(DH) DEHENEFFE, HAINAUT, HENNEBERT, LECHARLIER, PAULUS
Système de conception et d'exploitation d'une base de données.
Publication de l'Institut d'Informatique de Namur, 1974

(DL) DELVAUX, Y.
Implémentation d'un langage de haut niveau de manipulation de bases de données.
Institut d'Informatique Namur, 1977 (mémoire)

(H1) HAINAUT, J.L.
Un Modèle Descriptif de Bases de Données au Niveau Organique: le Modèle d'Accès.
Institut d'Informatique, Namur, mars 1980

(H2) HAINAUT, J.L.
Notes de cours (1. et 2. licence).
le Modèle d'Accès.
Institut d'Informatique, Namur

(IBM1) IMS/VS Primer Guide.

(IBM2) IMS/VS and DL/I DOS/VS Applications Analysis and Design

(IBM3) DB/DC Data Dictionary User's Guide

(IBM4) Techniques de base DL/I

(LE) LECHARLIER, B.
Séminaire de Programmation: Conception d'un Interpréteur
d'un Langage Simple et Didactique (1. Licence).
Institut d'Informatique, Namur.

(LF) LEFKOVITS, H.C.
Data Dictionary Systems
Q.E.D. Information Sciences,
Inc. Wellesley, Massachussets, 1977

(MA) MARTIN, J.L.
L'Informatique sans Programmeurs.
(traduit de l'américain par Partricia Braillon)
Les Editions d'Organisation, 1983.

(ME) MEYER B., BAUDOIN C.
Méthodes de programmation
Collection de la Direction des Etudes et Recherches
d'Electricité de France
Editions Eyrolles Paris, 1980.

(NO) NORI K.V., HAMMANN U., JENSEN K., NAGELI H.H., JACOBI Ch.
Pascal - P Implementation Notes
extrait de Pascal - the language and its implementation
(chap 9)
Edited by D.W. Banon
John Wiley & Sons ltd, 1981

(WI) WIRTH N.
Pascal - S: A subset and its implementation
extrait de Pascal - the language and its implementation
(chap 12)
Edited by D.W. Banon
John Wiley & Sons ltd, 1981

ERRATA

p 3, ligne 4 :	"Ceux-ci permettent la mise en forme de résultats complexes avec
p 3, ligne 16 :	"on line" par génération intermédiaire de programmes
p 4, ligne 7 :	etc.) de façon déterministe."
p 26, ligne 29 :	tent de développer des structures sous-jacentes à une vue utili-
p 27, ligne 28 :	système devrait être capable de faciliter les modifications de la vue au niveau
p 29, ligne 22 :	3.5.2.1. L'interface entre le schéma conceptuel et le schéma
p 30, ligne 11 :	schéma d'implémentation.
p 42, dernière ligne :	d'interfaces simulant la machine MAG.
p 43, ligne 9 :	Les SGBD's cibles de ces transformations se classent dans l'état
p 49, ligne 17 :	nombre d'occurrences permis pour chacune des zones. La
p 52, ligne 26 :	expression entière pour chaque dimension, séparée
p 54, ligne 1 :	Un item peut être élémentaire (si aucun de ses fragments n'est significatif
p 61, ligne 20 :	L'instruction FOR
p 64, ligne 3 :	la variable <VAR> et on effectue l'instruction <instr>
p 69, ligne 25 :	règles que dans la section 7.5.2.2.
p 87, ligne 3 :	Pour représenter les relations entre les occurrences d'un DATABASE
p 87, ligne 17 :	d'hashing et de chaînage, avec un accès par pointeur aux segments
p 90, ligne 19 :	Get Unique : l'appel GU est utilisé pour accéder à un segment
p 91, ligne 21 :	des codes de commande et/ou des qualifications de champ.
p 91, ligne 25 :	des opérateurs booléens. Un statement de qualification peut être
p 93, ligne 9 :	d'application) dans la même région ou partition. Le programme reçoit
p 93, ligne 32 :	inconnue ou non prévue par le PROCOPT, ou par exemple l'adresse
p 93, ligne 33 :	de PCB n'est pas l'une de celles reçues par le programme), sinon

p 109, ligne 23 : dans les langages de

p 109, ligne 1 : — PSB-IN : cette commande accède à la librairie DL/I des PSB's,

p 128, ligne 26 : et trier cette liste sur l'ordre demandé. Ensuite, la boucle

p 131, ligne 1 : la limitation

p 134, ligne 3 : — le numéro de la BD physique du type d'article,

p 141, ligne 12 : Le choix du DADIC nous a obligé à abandonner le caractère universel que nous

p 141, ligne 17 : algorithmique du rapport et une version du générateur sont implémentées. Cependant

p 141, ligne 26 : est judicieux pour une interface 'User Friendly'. Ce choix s'est justifié par la

Signification des abréviations

LDB — base de données logique.

Bibliographie

(BO) BOUTEL J-S, PIGNON J-P
Problématique et typologie des nouveaux langages de l'informatique.
Zéro. un. informatique mensuel (n° 178) Avril 1984

Facultés Universitaires Notre Dame de la Paix, Namur

Institut d'Informatique

Année académique 1983-1984

ETUDE D'UN GENERATEUR DE RAPPORTS SUR BASES DE DONNEES IMS

(annexes)

Marc GUEBELS
Philippe LANSSENS

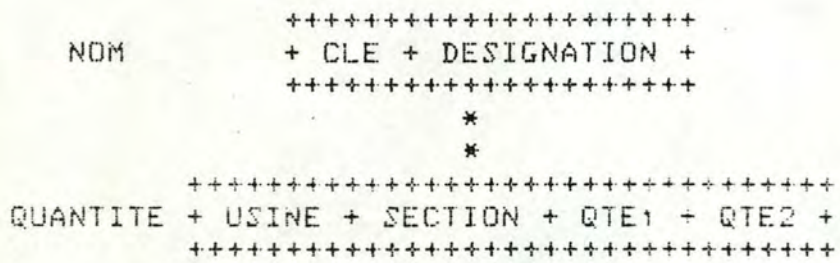
Mémoire présenté en vue de l'obtention du grade de liciencié et maître en informatique.

exemple du LEDGR et du LDL

Soit une base de données PRODUIT contenant 2 types d'article:

- NOM qui représente l'existence d'un produit particulier et qui possède 2 items CLE et DESIGNATION

- QUANTITE représentant les quantités produites dans chaque section de chaque usine pour un produit; il possède les items USINE, SECTION, QTE1 et QTE2. Les articles QUANTITE sont reliées à l'article NOM correspondant.



On veut obtenir pour les produits dont la clé est comprise entre deux valeurs données en paramétre, les quantités regroupées par produit et par usine ainsi que les totaux intermédiaires.

Pour obtenir ce rapport, il est nécéssaire de définir son format de sortie au moyen du LDL. Normalement cela devrait etre effectué au moyen d'un éditeur spécialisé. Celui-ci permettrait de visualiser le format d'une manière analogue à la figure 1.

Le rapport obtenu posséde une structuration hiérarchique donnée dans la figure 2 et consiste en 1 ensemble de zones décritesdans la figure 3.

Après avoir défini le format, il faut rédiger un programme qui accéde à la base de donnée et manipule les zones.
Un exemple en est donné à la figure 4.

FIGURE 1 : VISUALISATION AVEC ENCADREMENT DES ZONES D'UN EXEMPLE

LEGENDE :

LES "#" DELIMITENT LES ZONES.
LE NOM DE CHAQUE ZONE EST COMPRIS ENTRE "(" ET ")".
LES LIGNES ET LES COLONNES BLANCHES SONT PRéSENTES POUR GARDER L'ALIGNEMENT MALGRE LES ENCADREMENTS DES ZONES.
LES PARTIES COMPRISES ENTRE "<" ET ">" DONNENT LE NOM DE ZONES VARIABLES.

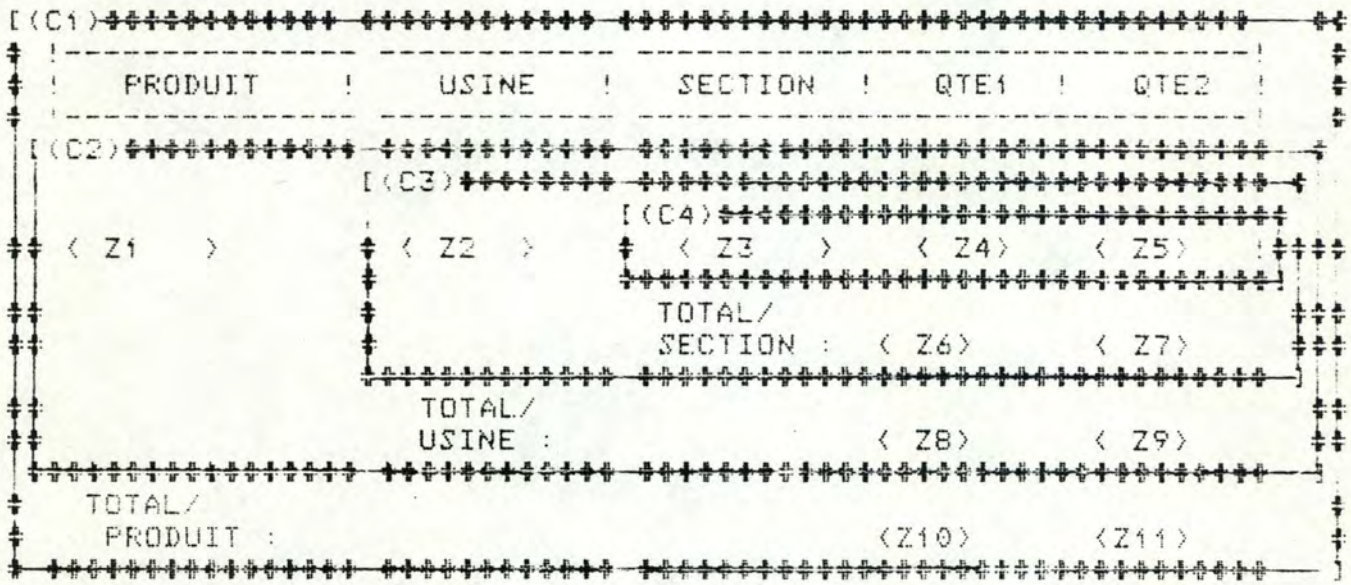


FIGURE 2 : STRUCTURATION HIERARCHIQUE DES ZONES

```
NIV 01 01 C1
          NIV 02 01 'PRODUIT'
                 02 'USINE'
                 03 'SECTION'
                 04 'QTE1'
                 05 'QTE2'
                 06 C2
                      NIV 03 01 Z1
                             02 C3
                                  NIV 04 01 Z2
                                         02 C4
                                              NIV 05 01 Z3
                                                     02 Z4
                                                     03 Z5
                                         03 'TOTAL/SECTION'
                                         04 Z6
                                         05 Z7
                             03 'TOTAL/USINE'
                             04 Z8
                             05 Z9
                 07 'TOTAL/PRODUIT'
                 08 Z10
                 09 Z11
```

FIGURE 3: DESCRIPTION COMPLETE DES ZONES

```
NOM INTERNE :01 00 01                ! NOM INTERNE :02 01 01
NOM EXTERNE :C1                      ! NOM EXTERNE :
POS VERT    :1                       ! POS VERT    :1
POS HOR     :1                       ! POS HOR     :1
REP MIN     :1                       ! REP MIN     :1
REP MAX     :1                       ! REP MAX     :1
TYPE GEO    :RECTANGLE               ! TYPE GEO    :RECTANGLE
HAUTEUR     :10                      ! HAUTEUR     :3
LONGUEUR    :60                      ! LONGUEUR    :16
ROLE        :ENSEMBLE                ! ENCADRE     : Y
                                     ! ROLE        :CONSTANTE
                                     ! VALEUR      : 'PRODUIT'
=============================================================================
NOM INTERNE :02 01 02                ! NOM INTERNE :02 01 03
NOM EXTERNE :                        ! NOM EXTERNE :
POS VERT    :1                       ! POS VERT    :1
POS HOR     :16                      ! POS HOR     :28
REP MIN     :1                       ! REP MIN     :1
REP MAX     :1                       ! REP MAX     :1
TYPE GEO    :RECTANGLE               ! TYPE GEO    :RECTANGLE
HAUTEUR     :3                       ! HAUTEUR     :3
LONGUEUR    :13                      ! LONGUEUR    :13
ENCADRE     : Y                      ! ENCADRE     : Y
ROLE        :CONSTANTE               ! ROLE        :CONSTANTE
VALEUR      : 'USINE'                ! VALEUR      : 'SECTION'
=============================================================================
NOM INTERNE :02 01 04                ! NOM INTERNE :02 01 05
NOM EXTERNE :                        ! NOM EXTERNE :
POS VERT    :1                       ! POS VERT    :R 1
POS HOR     :40                      ! POS HOR     :R 50
REP MIN     :1                       ! REP MIN     :1
REP MAX     :1                       ! REP MAX     :1
TYPE GEO    :RECTANGLE               ! TYPE GEO    :RECTANGLE
HAUTEUR     :3                       ! HAUTEUR     :3
LONGUEUR    :11                      ! LONGUEUR    :11
ENCADRE     : Y                      ! ENCADRE     : Y
ROLE        :CONSTANTE               ! ROLE        :CONSTANTE
VALEUR      : 'QTE1'                 ! VALEUR      : 'QTE2'
=============================================================================
NOM INTERNE :02 01 06                ! NOM INTERNE :03 06 01
NOM EXTERNE :C2                      ! NOM EXTERNE :Z1
POS VERT    :1                       ! POS VERT    :1
POS HOR     :4                       ! POS HOR     :1
REP MIN     :1                       ! REP MIN     :1
REP MAX     :*                       ! REP MAX     :1
TYPE GEO    :RECTANGLE               ! TYPE GEO    :RECTANGLE
HAUTEUR     :5                       ! HAUTEUR     :1
LONGUEUR    :60                      ! LONGUEUR    :8
ROLE        :ENSEMBLE                ! ROLE        :VARIABLE
                                     ! FORMAT      : X(8)
```

```
NOM INTERNE :03 06 02                ! NOM INTERNE :04 02 01
NOM EXTERNE :C3                      ! NOM EXTERNE :Z2
POS VERT    :1                       ! POS VERT    :1
POS HOR     :17                      ! POS HOR     :1
REP MIN     :1                       ! REP MIN     :1
REP MAX     :*                       ! REP MAX     :1
TYPE GEO    :RECTANGLE               ! TYPE GEO    :RECTANGLE
HAUTEUR     :3                       ! HAUTEUR     :1
LONGUEUR    :44                      ! LONGUEUR    :8
ROLE        :ENSEMBLE                ! ROLE        :VARIABLE
                                     ! FORMAT      : X(8)
=============================================================================
NOM INTERNE :04 02 02                ! NOM INTERNE :05 02 01
NOM EXTERNE :C4                      ! NOM EXTERNE :Z3
POS VERT    :1                       ! POS VERT    :1
POS HOR     :13                      ! POS HOR     :1
REP MIN     :1                       ! REP MIN     :1
REP MAX     :1                       ! REP MAX     :1
TYPE GEO    :RECTANGLE               ! TYPE GEO    :RECTANGLE
HAUTEUR     :1                       ! HAUTEUR     :1
LONGUEUR    :32                      ! LONGUEUR    :8
ROLE        :ENSEMBLE                ! ROLE        :VARIABLE
                                     ! FORMAT      : X(8)
=============================================================================
NOM INTERNE :05 02 02                ! NOM INTERNE :05 02 03
NOM EXTERNE :Z4                      ! NOM EXTERNE :Z5
POS VERT    :1                       ! POS VERT    :1
POS HOR     :13                      ! POS HOR     :23
REP MIN     :1                       ! REP MIN     :1
REP MAX     :1                       ! REP MAX     :1
TYPE GEO    :RECTANGLE               ! TYPE GEO    :RECTANGLE
HAUTEUR     :1                       ! HAUTEUR     :1
LONGUEUR    :8                       ! LONGUEUR    :8
ROLE        :VARIABLE                ! ROLE        :VARIABLE
                                     ! FORMAT      : X(8)
=============================================================================
NOM INTERNE :04 02 03                ! NOM INTERNE :04 02 04
NOM EXTERNE :                        ! NOM EXTERNE :Z6
POS VERT    :2                       ! POS VERT    :3
POS HOR     :14                      ! POS HOR     :25
REP MIN     :1                       ! REP MIN     :1
REP MAX     :1                       ! REP MAX     :1
TYPE GEO    :RECTANGLE               ! TYPE GEO    :RECTANGLE
HAUTEUR     :2                       ! HAUTEUR     :1
LONGUEUR    :10                      ! LONGUEUR    :8
ROLE        :CONSTANTE               ! ROLE        :VARIABLE
VALEUR      :'TOTAL/SECTION : '      ! FORMAT      : X(8)
```

```
NOM INTERNE  :04 02 05          ! NOM INTERNE  :03 06 03
NOM EXTERNE  :Z7                ! NOM EXTERNE  :
POS VERT     :3                 ! POS VERT     :4
POS HOR      :35                ! POS HOR      :19
REP MIN      :1                 ! REP MIN      :1
REP MAX      :1                 ! REP MAX      :1
TYPE GEO     :RECTANGLE         ! TYPE GEO     :RECTANGLE
HAUTEUR      :1                 ! HAUTEUR      :2
LONGUEUR     :8                 ! LONGUEUR     :10
ROLE         :VARIABLE          ! ROLE         :CONSTANTE
FORMAT       : X(8)             ! VALEUR       :'TOTAL/USINE :'
=================================================================================
NOM INTERNE  :03 06 04          ! NOM INTERNE  :03 06 05
NOM EXTERNE  :Z8                ! NOM EXTERNE  :Z9
POS VERT     :5                 ! POS VERT     :5
POS HOR      :41                ! POS HOR      :61
REP MIN      :1                 ! REP MIN      :1
REP MAX      :1                 ! REP MAX      :1
TYPE GEO     :RECTANGLE         ! TYPE GEO     :RECTANGLE
HAUTEUR      :1                 ! HAUTEUR      :1
LONGUEUR     :8                 ! LONGUEUR     :8
ROLE         :VARIABLE          ! ROLE         :VARIABLE
FORMAT       : X(8)             ! FORMAT       : X(8)
=================================================================================
NOM INTERNE  :02 01 07          ! NOM INTERNE  :02 01 08
NOM EXTERNE  :                  ! NOM EXTERNE  :Z10
POS VERT     :9                 ! POS VERT     :10
POS HOR      :3                 ! POS HOR      :41
REP MIN      :1                 ! REP MIN      :1
REP MAX      :1                 ! REP MAX      :1
TYPE GEO     :RECTANGLE         ! TYPE GEO     :RECTANGLE
HAUTEUR      :2                 ! HAUTEUR      :1
LONGUEUR     :10                ! LONGUEUR     :8
ROLE         :CONSTANTE         ! ROLE         :VARIABLE
VALEUR       :'TOTAL/ PRODUIT :' ! FORMAT       : X(8)
=================================================================================
NOM INTERNE  :02 01 09
NOM EXTERNE  :Z11
POS VERT     :10
POS HOR      :51
REP MIN      :1
REP MAX      :1
TYPE GEO     :RECTANGLE
HAUTEUR      :1
LONGUEUR     :8
ROLE         :VARIABLE
FORMAT       : X(8)
```

FIGURE 4 : PROGRAMME LEDGR

```
RAPPORT1: PROCEDURE;
 DECLARE DATABASE PRODUIT;
 DECLARE RAPPORT RAPPORT1;
 DECLARE PAR%1 PARAMETER CHAR;
 DECLARE PAR%2 PARAMETER CHAR;

 Z_NEW(C1);
 FOR_EACH N=NOM WHERE((CLE )>= PAR%1)AND(CLE (<= PAR%2)) DO
     Z_NEW(C2);
     Z1 = N.DESIGNATION;
     FOR_EACH Q1=QUANTITE FROM N GROUPED BY USINE DO
       Z_NEW(C3);
       Z2 = Q1.USINE;
       FOR_EACH Q2=QUANTITE FROM Q1 DO
         Z_NEW(C4);
         Z3 = Q2.SECTION;
         Z4 = Q2.QTE1;
         Z5 = Q2.QTE2;
         Z-END(C4);
         END;
       Z6 = SUM(Q2.QTE1,Q2);
       Z7 = SUM(Q2.QTE2,Q2);
       Z-END(C3);
       END;
     Z8 = SUM(Q2.QTE1,Q1);
     Z9 = SUM(Q2.QTE2,Q1);
     Z-END(C2);
     END;
 Z10 = SUM(Q2.QTE1,N);
 Z11 = SUM(Q2.QTE2,N);
 END-FOR;
 Z-END(C1);
END;
```

DEFINITION DE L'INTERFACE MAG-IMS

1) INTRODUCTION

L'interface que nous avons réalisée permet d'accéder à un ensemble de bases de données IMS indépendamment des problèmes de positionnement et de gestion des PCB's.

Comme l'appel à DLI se fait par passage de paramètres, il est possible de concevoir une interface pour toute base de données. L'interface doit cependant avoir un minimum d'information sur les bases de données qu'il va devoir gérer. Celles-ci lui seront fournies par le programme utilisant l'interface lors du premier appel.

L'interface n'implémente pas tous les concepts du MAG:

- Vu que nous n'avons besoin que d'accéder en lecture aux bases de données, seules des primitives d'acces sont définies.

- Seul l'accès aux articles d'un type est envisagé.

- Tout accès est effectué dans un chemin:
  chemin implicite au travers de toute la base de données,
  chemin imlicite par hiérarchie physique,
  chemin explicite par lien logique.

- Seul l'accès au premier et à l'article suivant d'un chemin est défini.

- L'accès au valeurs d'item est effectué automatiquement. En effet, IMS ne fait pas de difference entre l'accès aux articles et l'accès aux valeurs d'item.

- Il n'y a pas de primitives spéciales pour l'accès par clé.

- Seul l'ordre défini par l'index primaire est utilise pour le moment.

2) APPEL A L'INTERFACE

Il existe 2 points d'entrée pour l'interface
- un premier pour initialiser l'interface;
- un second pour accéder à un article

2.1) INITIALISATION DE L'INTERFACE

On passe les adresses de 4 tables.
la table de description des BD's
la table de description des segments
la table de description des PCB's
la table des numéros d'opérations

* Table des DB's

La table des DB contient une entree pour chaque DB IMS accessible.
Chaque élément de la table des DB's contient
- le nom de la DB (à titre informatif);
- des références dans la table des PCB pour les PCB qui permettent d'accéder à la DB;
- une référence dans la table des segments, qui correspond au type de segment racine de la DB;
- le nombre de type de segment appartenant à la DB.

```
01 DB(*),                              /*    TABLE DES DB              */
   02 DB_NAME       CHAR(8),           /* NOM IMS DE LA DB             */
   02 PCB(2)        BIN FIXED(15),     /* REFERENCE VERS TABLE PCB     */
   02 P_SEGM        BIN FIXED(15),     /* REFERENCE VERS TABLE SEGM    */
   02 NB_SEGM       BIN FIXED(15),     /* PREMIER ET NOMBRE ENTREES    */
   02 RESERVE       CHAR(2);           /*                              */
```

* Table des segments

La table des segments contient la description de tous les types de segments des DB's, dans leur ordre dans la DB. Chaque entrée contient les informations suivantes:

- le nom IMS du segment,
- le numéro de niveau,
- le numéro du parent,
- les positions et longueurs des clés de séquence et concaténées,

Si un segment ne posséde pas de clé de séquence, il ne possède pas d'identifiant. On ne peut pas se repositionner et on obtiendra toujours le meme segment si on veut accéder à une séquence. Si l'interface le détecte, il positionnera un code d'erreur.

- la zone réservée peut dans certains cas contenir le nom d'un champ jouant un role particulier pour le segment.

```
01 SEGM(*),                          /*     TABLE DES SEGMENTS      */
   02 SEGM_NAME     CHAR(8),         /* NOM IMS DU SEGMENT          */
   02 NUM_PAR       BIN FIXED(15),   /* NUMERO DU PARENT            */
   02 NUM_NIV       BIN FIXED(15),   /* NUMERO DE NIVEAU            */
   02 SEGM_LH       BIN FIXED(15),   /* LONGUEUR SEGMENT            */
   02 KEY_LH        BIN FIXED(15),   /* LONGUEUR CLE SEQUENCE       */
   02 KEY_POS       BIN FIXED(15),   /* POSITION CLE SEQUENCE       */
   02 CKEY_LH       BIN FIXED(15),   /* LONGUEUR CLE CONCATENNEE    */
   02 RESERVE       CHAR(8);         /*                             */
```

* Table des PCB

La table des PCB contient pour chaque PCB, l'adresse communiquée par IMS lors de l'appel du programme.
En plus elle contient la référence vers la table des numéros d'opération correspondant au type de segment racine du PCB.

```
01 PCB(*)                           /*     TABLE DES PCB           */
   02 PCB_PTR       PTR,             /* POINTEUR VERS PCB           */
   02 P_SEGM_OPER   BIN FIXED(15),   /* PREMIER NUMERO OPERATION    */
   02 NB_SEGM       BIN FIXED(15),   /* NOMBRE SEGMENT              */
   02 RESERVE       CHAR(2),         /*                             */
```

* Table des numéros d'opération

La table des numéros d'opération contient une entrée pour chaque type de segment accessible dans chaque PCB. Elle permet de mémoriser le numéro d'opération qui a effectué un positionnement sur le type de segment de ce PCB.
A chaque exécution d'une boucle, on attribue un nouveau numéro d'opération qui est stocké dans la variable d'article;
Lors d'un accès qui demande un positionnement, on se sert de ces numéros pour vérifier si le positionnement est encore valable.

```
01 NUM_OPER(*)    BIN FIXED(15); /* NUMERO OPERATION              */
```

2.1) ACCES AUX ARTICLES

Pour accéder aux articles on spécifie les paramètres suivants

- COOP
- VARIABLE_CIBLE
- VARIABLE_SOURCE
- TYPE_CHEMIN
- CONDITION
- CODE_RETOUR

* COOP

COOP spécifie le type d'opération que l'on veut effectuer:

```
01 COOP,                          OU   COOP CHAR(5)
   02 TYPE      CHAR(1),
   02 POSITION  CHAR(1),
   02 QUALIF    CHAR(1),
   02 CHEMIN    CHAR(1),
   02 RESERVE   CHAR(1);
```

où :

- type = 'A' pour un accès (seul implémenté)

- position = 'P' pour l'accès au premier
  'S' suivant

- qualif = 'Q' pour un accès avec conditions
  'N' sans condition

- chemin = 'C' pour un accès dans un chemin inter-article
  'B' dans toute la base de données.

* VARIABLE_CIBLE et VARIABLE_SOURCE

VARIABLE_CIBLE et VARIABLE_SOURCE sont des variables d'article identifiant l'article source et l'article cible.
Si on désire accéder aux articles de la DB cible d'un chemin, on spécifie l'origine du chemin dans VARIABLE_SOURCE.

Une variable d'article contient:

- le numéro du type d'article; il s'agit du numéro local à une DB.
- le numéro de la DB auquel il appartient.
  Ces deux numéros sont utilisés pour accéder aux tables de description.
- l'adresse de la zone de stockage des valeurs d'item
- la place pour stocker la clé concaténée
- le numéro d'opération

* TYPE_CHEMIN

TYPE_CHEMIN spécifie le chemin à utiliser.
Si on utilise un chemin hiérachique physique, il faut spécifier une valeur nulle.
Si on utilise une relation logique, on indique la référence du segment enfant logique par lequel la relation est implémentée.

```
01 TYPE_CHEMIN
  02 NUM_DB          BIN FIXED(15),
  02 NUM_SEGM        BIN FIXED(15);
```

* CONDITION

CONDITION est une zone qui contient des qualifications sur les valeurs des champs. Cette zone doit etre au format IMS:

```
01 CONDITION CHAR(*)
```

* CODE_RETOUR

CODE_RETOUR indique la maniere dont l'opération s'est déroulée.

```
01 CODE_RETOUR   BIN FIXED(15);
```

3) MODE DE FONCTIONNEMENT DE L'INTERFACE

3.1) SCHEMA GENERAL

Pour effectuer un accès, on suivra ces étapes:

1) Choix du PCB sur lequel on va effectuer l'ordre:
Quand il y a plusieurs PCB disponibles pour accéder a une DB, l'interface en choisit un qui lui évitera de perdre la position courante dans les boucles ou dans les cheminements hiérarchiques.

2) Repositionnement éventuel:
L'accès dans un chemin d'accès se faisant par rapport à la position d'un article source du chemin, il faut éventuellement se repositionner si on a perdu la position de ce dernier suite à un accès effectué entretemps.
De meme, lors de l'accès au suivant d'un article dans un chemin , on peut avoir perdu la position.

3) Acces proprement-dit;

4) test du code-retour IMS
mise-à-jour du descripteur de segment,
de la zone de réception des valeurs d'item,
des tables de mémorisation du positionnement;

3.2) DESCRIPTION DES DIFFERENTS ACCES

Il y a 4 grands types d'accès
- accès au premier article dans toute la base de donnée
- accès à l'article suivant dans toute la base de données
- accès au premier article cible d'un chemin inter-article
- accès à l'article suivant cible d'un chemin inter-article

3.2.1) ACCES PREMIER BASE DE DONNEES

Explication :

L'accès se fait par un GET UNIQUE sur le type de segment cible.
Il ne faut pas se positionner;
Si accès réussi,
on est positionné sur le segment cible, sur ses parents ainsi que sur le premier des descendants éventuels dans chaque type de segment.
Si echec,
position inconnue.

Pseudo code :

- DLI CODE = GU
PCB = LIBRE CHOIX
SSA1
NAME = CIBLE
CF =
COND = COND SUR CIBLE

3.2.2) ACCES SUIVANT BASE DE DONNEES

Explication :

Il faut vérifier s'il faut se repositionner sur le segment courant
Si oui, cela s'effectue par un GU avec code de fonction 'C' et la clé concaténée du segment cible courant.
On utilise un GN pour accéder au suivant.
Si accès réussi,
    On est positionné sur le segment cible, sur ses parents ainsi que sur le premier des descendants éventuels dans chaque type de segment.
Si échec,
    position inconnue.

Pseudo code :

```
- TEST  POSITIONNEMENT CIBLE
  SI PERDU
  DLI  CODE   = GU
       PCB    = LIBRE CHOIX
       SSA1
        NAME  = CIBLE
        CF    = C
        COND  = CLE CIBLE

- DLI  CODE   = GN
       PCB    = LE MEME QUE CELUI SUR LEQUEL ON EST POSITIONNE
       SSA1
        NAME  = CIBLE
        CF    =
        COND  = COND SUR CIBLE
```

3.2.3) ACCES PREMIER CHEMIN

Explication :

3 cas sont à envisager suivant que le chemin demande l'accès
a) d'un parent physique vers un enfant physique,
b) d'un enfant physique vers son parent physique,
c) d'un segment vers un autre par un lien logique.

On se trouve dans le cas a) ou b) si le nom du chemin n'est pas spécifié (ou valeur nulle). La différence entre le cas a) et le cas b) se fait par comparaison des numéros de niveau des types de segment source et cible

On reconnait que l'on est dans le cas c) si le nom du chemin correspond à un type de segment enfant logique.

A) ACCES PAR CHEMIN PHYSIQUE

Explication :

Il faut éventuellement se repositionner sur le segment source
Si oui, cela s'effectue par un GU avec code de fonction 'C' et la clé concaténée du segment source courant.
On utilise un GN avec deux SSA; un sur le segment source avec code de fonction V et un sur le segment cible avec code de fonction F et la condition sur la cible.
Si accès réussi,
on est positionné sur le segment cible, sur ses parents ainsi que sur le premier des descendants éventuels dans chaque type de segment.
Si echec,
position inconnue.

Pseudo code

```
- TEST  POSITIONNEMENT DE LA SOURCE
  SI PERDU
  DLI   CODE    = GU
        PCB     = LIBRE CHOIX
        SSA1
         NAME   = SOURCE
         CF     = C
         COND   = CLE SOURCE

- DLI   CODE    = GN
        PCB     = LE MEME QUE CELUI SUR LEQUEL ON EST POSITIONNE
        SSA1
         NAME   = SOURCE
         CF     = V
         COND   =
        SSA2
         NAME   = CIBLE
         CF     = F
         COND   = COND SUR CIBLE
```

B) ACCES PAR CHEMIN PHYSIQUE INVERSE

Explication :

IL faut éventuellement se repositionner sur le segment source (c'est à dire l'enfant)
Si oui, cela s'effectue par un GU avec code de fonction 'C' et la clé concaténée du segment source courant.
On utilise un GU avec code de fonction V pour accéder à la cible
L'accès ne peut etre que réussi, puiqu'un segment enfant possède toujours un parent physique.

On est positionné sur le segment cible, sur ses parents ainsi que sur le premier des descendants éventuels dans chaque type de segment.

NB: un autre algorithme possible consiste à récupérer la clé du segment parent comme sous-clé du segment source.

Pseudo code :

```
- TEST  POSITIONNEMENT DE LA SOURCE
  SI PERDU
  DLI  CODE   = GU
       PCB    = LIBRE CHOIX
       SSA1
        NAME  = SOURCE
        CF    = C
        COND  = CLE SOURCE
- DLI  CODE   = GU
       PCB    = LE MEME QUE CELUI SUR LEQUEL ON EST POSITIONNE
       SSA1
        NAME  = CIBLE
        CF    = V
        COND  = COND SUR CIBLE
```

C) ACCES PAR CHEMIN LOGIQUE

Explication :

L'accès par une relation logique va etre effectué en récupérant dans le segment enfant logique, la clé de son parent logique.
Il faut éventuellement se repositionner sur le segment source.
Si oui, cela s'effectue par un GU avec code de fonction 'C' et la clé concaténée du segment source courant.
On accède ensuite au segment enfant logique par un GN avec double SSA sur la source (code F) et sur l'enfant (code F)
Si on a réussi, on récupère la clé du parent logique et on utilise un GU avec code de fonction V pour accéder à la cible

La relation logique peut porter sur des segments appartenant à la même BD ou à des BD différentes.
Dans le premier cas, si on emploie le même PCB pour accéder au segment source et au segment cible, lors de l'accès au segment cible, on perd d'office le positionnement sur le segment source.
Pour optimiser, s'il existe 2 PCB pour la même DB, on essaiera d'en utiliser un pour l'accès au segment source et au segment enfant logique tandis que l'autre servira pour l'accès à la cible.

Puisqu'on accède au segment cible par un GU avec code de fonction C, il n'est pas possible de spécifier de conditions sur les valeurs d'item.
Par contre on peut envisager de spécifier des conditions sur le segment enfant logique. Dans le cas où une condition est exprimée, c'est lors de l'accès à l'enfant logique qu'elle sera appliquée.

Pseudo code :

```
- TEST  POSITIONNEMENT DE LA SOURCE
  SI PERDU
  DLI  CODE    = GU
       PCB     = LIBRE CHOIX
       SSA1
        NAME   = SOURCE
        CF     = C
        COND   = CLE SOURCE
- DLI  CODE    = GN
       PCB     = LE MEME QUE CELUI SUR LEQUEL ON EST POSITIONNE
       SSA1
        NAME   = SOURCE
        CF     = V
        COND   =
       SSA2
        NAME   = ENFANT LOGIQUE
        CF     = F
        COND   = COND SUR CIBLE
- TEST SI REUSSI
  DLI  CODE    = GU
       PCB     = LIBRE CHOIX
       SSA1
        NAME   = CIBLE
        CF     = C
        COND   = CLE DANS ENFANT LOGIQUE
```

3.2.4) ACCES SUIVANT CHEMIN

Explication :

globalement on va retrouver la même situation qu'en 3.2.3

A) ACCES PAR CHEMIN PHYSIQUE

Explication :

Il faut éventuellement se repositionner sur les segments source et cible courants.
Si oui, cela s'effectue par un GU sur le segment cible avec code de fonction C.
On utilise un GN avec 2 SSA pour accéder à la cible suivante.
Si accès réussi,
    on est positionné sur le segment cible, sur ses parents ainsi que sur le premier des descendants éventuels dans chaque type de segment.
Si echec
    position inconnue.

Pseudo code :

```
- TEST  POSITIONNEMENT DE LA SOURCE
  SI PERDU
  DLI   CODE    = GU
        PCB     = LIBRE CHOIX
        SSA1
         NAME   = CIBLE
         CF     = C
         COND   = CLE CIBLE
- DLI   CODE    = GN
        PCB     = LE MEME QUE CELUI SUR LEQUEL ON EST POSITIONNE
        SSA1
         NAME   = SOURCE
         CF     = V
         COND   =
        SSA2
         NAME   = CIBLE
         CF     =
         COND   = COND SUR CIBLE
```

B) ACCES PAR CHEMIN PHYSIQUE INVERSE

Explication :

L'accès est nécessairement un échec puisqu'un segment ne peut avoir qu'un seul parent physique.

C) ACCES PAR CHEMIN LOGIQUE

Explication :

Il faut éventuellement se repositionner sur le segment source et le segment enfant logique auquel correspond le segment cible. Ici il peut y avoir 2 solutions:
Soit le segment enfant logique possède une zone de séquence qui soit la clé du parent logique. Dans ce cas sa clé concaténée est formée par la clé de son parent physique et de son parent logique, et on peut utiliser un SSA avec code de fonction C.
Soit le segment ne possède pas de clé de séquence ou une clé de séquence différente de la clé du parent logique. Dans ce cas il faut utiliser un SSA avec une qualification sur le champ qui contient la clé du parent logique.
Une fois positionné, il faut accéder au segment enfant logique suivant et puis au segment parent logique.

Pseudo code :

```
- TEST  POSITIONNEMENT DE L'ENFANT LOGIQUE
  SI PERDU
  PREMIER CAS:
  DLI  CODE    = GU
       PCB     =
       SSA1
        NAME   = ENFANT LOGIQUE
        CF     = C
        COND   = CLE SOURCE+CLE CIBLE
  DEUXIEME CAS:
  DLI  CODE    = GU
       PCB     = LIBRE CHOIX
       SSA1
        NAME   = SOURCE
        CF     = C
        COND   = CLE SOURCE
- DLI  CODE    = GN
       PCB     =
       SSA1
        NAME   = SOURCE
        CF     = V
        COND   =
       SSA2
        NAME   = ENFANT LOGIQUE
        CF     =
        COND   =
         FIELDNAME = NOM DU CHAMP
         CO        = '= '
         VALEUR    = CLE DU PARENT LOGIQUE
```

```
- DLI   CODE    = GN
        PCB     =
        SSA1
         NAME   = SOURCE
         CF     = V
         COND   =
        SSA2
         NAME   = ENFANT LOGIQUE
         CF     =
         COND   = COND SUR ENFANT LOGIQUE
- TEST SI REUSSI
  DLI   CODE    = GU
        PCB     = LIBRE CHOIX
        SSA1
         NAME   = CIBLE
         CF     = C
         COND   = CLE DANS ENFANT LOGIQUE
```

Annexe au chapitre de présentation du SGBD IMS

Notre volonté dans cette annexe n'est pas d'illustrer les concepts généraux des bases de données, mais uniquement les notions propres aux bases de données hiérarchiques IMS.

Nous allons prendre un exemple classique et supposer qu'un organisme bancaire veuille structurer certaines informations concernant trois services qu'il apporte à ses clients : la gestion de comptes chèques, la vente de titres et l'octroi de prêts. Cet exemple est tiré en grande partie du cours 'Techniques de Base DL1' d'IBM.

Sur la gestion de comptes chèques, il désire identifier le propriétaire de chaque compte, le compte chèque en lui-meme ainsi que les différents mouvements que le compte subit. Pour la vente des titres, il veut une identification de chaque acheteur de titre, ainsi que le titre acheté. Les informations à retenir en ce qui concerne l'octroi d'un pret est la certification de l'emprunteur, le prêt en lui-même, ainsi que les différents remboursements déjà reçus par la banque.

Pour représenter les informations sur ces trois services en une base de données de type hiérarchique, il faut d'abord trouver un élément commun et central à ces trois services. C'est sur cet élément central qu'est articulée la base de données. Dans notre exemple, les trois services concernent une même population et nécessitent une identification. Aussi, celle-ci, symbolisée par ID, est notre point central. Dans certains cas, les titulaires d'un compte, les propriétaires d'un titre, les bénéficiaires d'un pret sont les mêmes personnes et donc, de même ID. Une même personne peut être titulaire de plusieurs comptes, propriétaires de différents titres et bénéficiaires de différents prêts.

Par rapport à l'axe ID, les informations compte chèque (symbolisée par CHEQUE), titre (TITRE) et prêt (PRET) se trouvent dans la même situation. Ce sont des informations qui se rattachent directement sous l'élément central. La structure est complète quand on a placé l'information de mouvement (MOUV) et de remboursement (REMB) respectivement en-dessous de CHEQUE et PRET. Elle est représentée par la figure IMS.1. Cette figure n'est qu'un dessin et nous allons définir ce qui se trouve derrière chaque rectangle et derrière les traits qui les relient. Supposons que cette banque, pour le besoin de l'exemple, ne possède que deux clients enregistrés dans la BD (fig. IMS.2).

Derrière chaque rectangle se trouve un part des données. Par exemple, derrière ID, nous retrouvons les clients DUPONT et MARTIN. Chacun d'eux occupe un espace distinct dont l'importance est choisi par le concepteur. Derrière CHEQUE se trouvent tous les comptes tenus par la banque. Chacun d'eux occupe également un espace distinct, qui peut être différent de celui d'ID. Il en va de même pour MOUV, TITRE, PRET et REMB.

Chaque type de donnée est ainsi isolé et nous pouvons y accéder de façon séparée, sur base de leur type.

Qu'y-a-t-il derrière les traits ? Mr Dupont est titulaire de de deux comptes chèques, propriétaires de deux titres et bénéficiaire d'un prêt. Aussi sur la figure IMS.2, nous pouvons voir partant de Dupont trois chaines de flèches qui lui associent ces éléments. Puis, chacun des comptes chèques de Dupont est relié à son unique remboursement. Mr Martin n'apparait que bénéficiaire d'un prêt. Ainsi, les données sont structurées ou en relation.

Nous avons une base de données où un élément ne peut apparaitre qu'une fois et où les données sont structurées, c'est-à-dire séparées, mais en relation.

De cette base de données, nous allons en définir les principaux composants. Chacune des informations apparaissant dans la structure sous la forme d'un grand rectangle s'appelle segment-type. C'est l'élément de base d'une structure hiérarchique. Cette structure a trois niveaux (fig IMS.1). Au premier niveau, nous ne pouvons placer qu'un seul segment type appelé racine. Tous les autres segments types sont des segments dépendants. Nous pouvons distinguer trois chemins. Le premier est constitué par ID, CHEQUE, MOUV. Le second par ID et TITRE. Le troisième par ID, PRET et REMB.

Les trois chemins, tous issus de la racine, ne peuvent se recouper. C'est la raison pour laquelle ce type de structure est dit en arbre.

Le segment type est une entité qu'on ne retrouve pas sur un support physique. Par contre, on y trouve des occurrences de segment type. Chacune d'elle a pour longueur la valeur associée au segment type au moment de sa définition. Lorsqu'un programme exploite une BD, il a donc à lire ou à écrire, modifier ou supprimer des occurrences de segment type. En DL1, l'occurrence représente l'unité logique d'entrée / sortie.

Prenons une occurrence de la racine (DUPONT). Un ensemble de flèches conduisent aux différents comptes; et de là, à leurs mouvements, aux différents titres, puis au prêt et de là, à son remboursement. Tout cet ensemble d'occurrences constitue un Data Base Record. On appelle donc data base record une occurrence de la racine et tous ses dépendants. La figure IMS.3 représente le data base record de racine DUPOND.

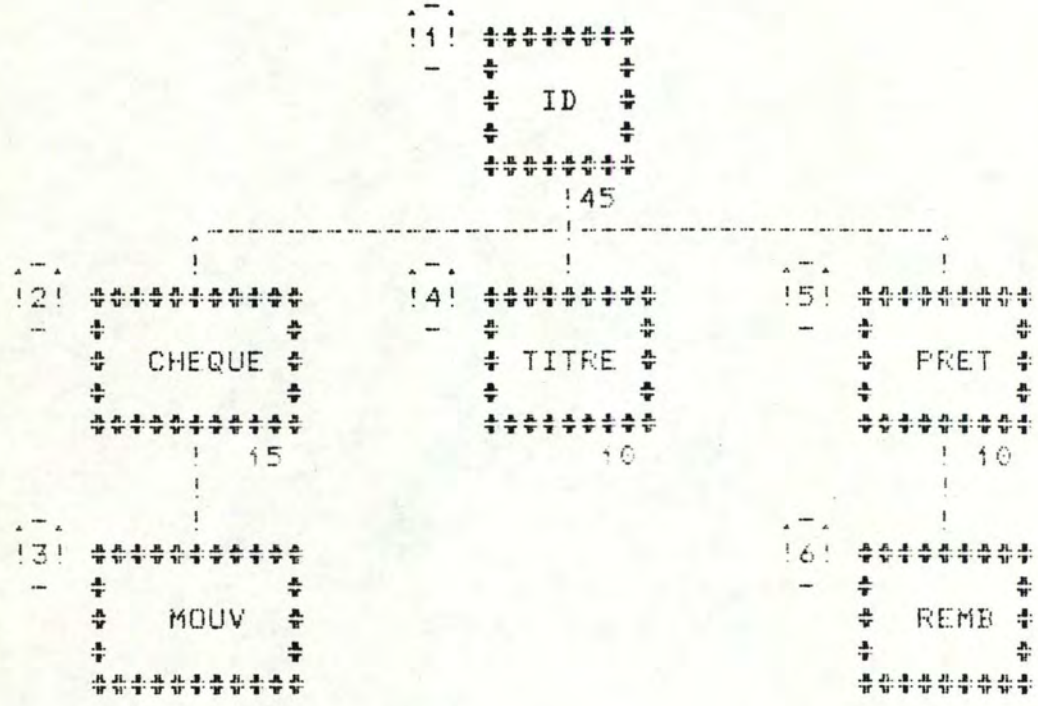


Fig. IMS.4

Sur la figure IMS.4, les numéros encadrés qui se trouvent à coté de la repésentation de chaque type de segment exprime l'ordre des seg ments selon la séquence hiérarchique.

DESCRIPTION DES BASES DE DONNEES

Celle-ci se fait par un jeu de macro-instructions qui permet d'une part de définir la structure de la BD dans le plus menu détail; d'autre part de déterminer les modalités de la réalisation physique de cette BD par DL1. Le résultat de cette opération est un bloc de controle, le DBD. Cette description étant séparée du programme, il sera assez facile de la réécrire pour changer le type d'organisation ou meme pour modifier la structure, sans aucune conséquence pour le programme.

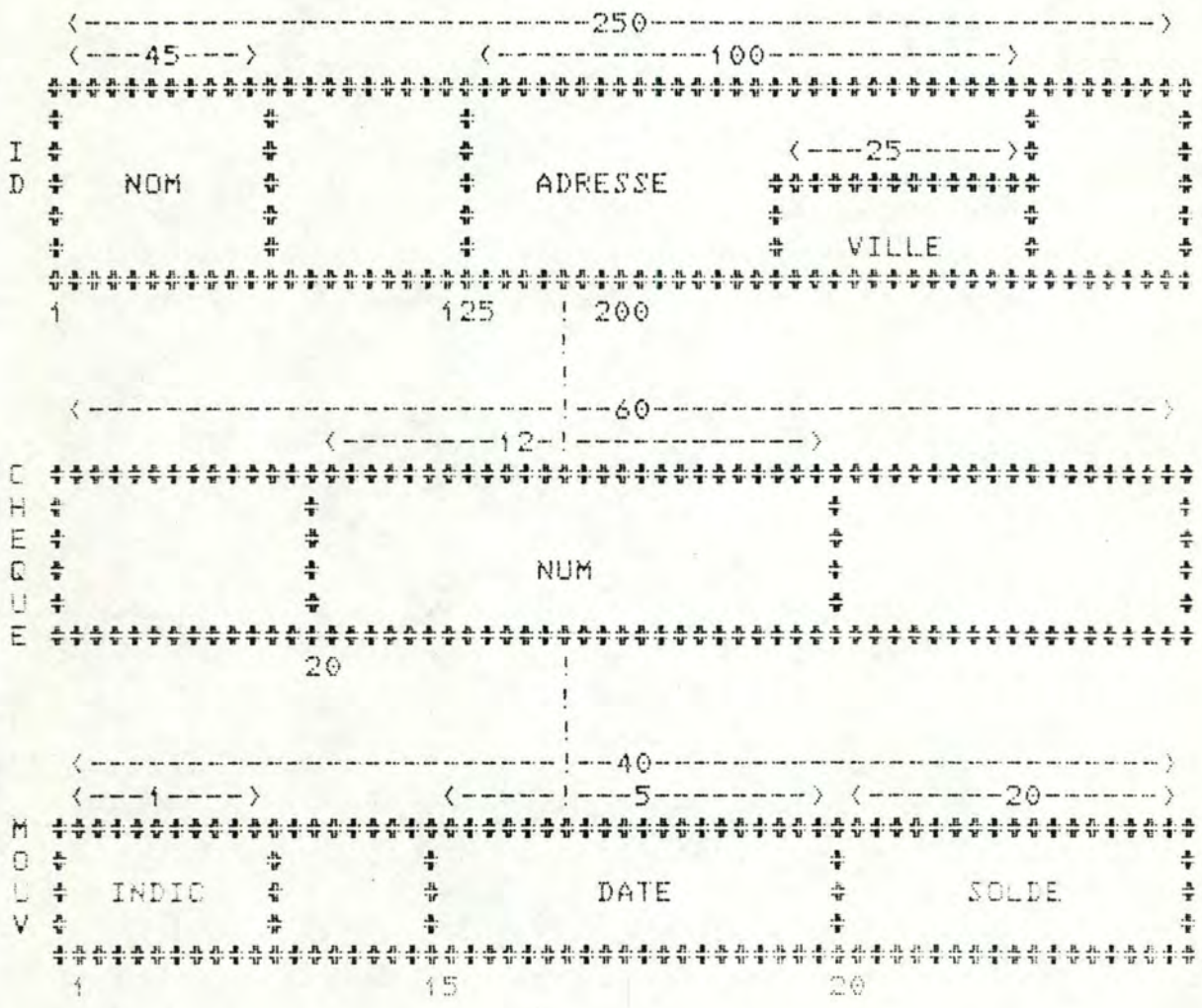


fig. IMS.5

Le DBD
------

1. dans le but de définir la structure des données

Chaque segment type sera décrit dans l'ordre hiérarchique. A chaque segment type correspond une macro-instuction SEGM. Il faut compléter l'énumération des segments types par un paramètre indiquant la position relative du segment à définir. Le paramètre PARENT définit le segment dont dépend directement le segment qu'on est en train de définir (paramètre qui n'est pas nécessaire pour la racine). Le paramètre BYTES notifie la longueur de chaque occurrence du segment type. Sur le figure IMS.5, nous retrouvons les champs définis sur les segments qui constituent le premier chemin de

la BD. Chaque macro-instruction SEGM peut être suivie d'une ou plusieurs instructions FIELD définissant chacune la définition d'un champ.

Un champ a toujours une position de départ (le paramètre START), une longueur (paramètre BYTES). Chaque segment peut avoir un champ privilégié toujours défini en tete, meme si d'autres champs le précèdent dans le segment. On l'appelle clé du segment et on le définit à l'aide du mot SEQ. Le champ clé est indispensable pour une racine, car DL1 s'en sert,

- pour batir l'index de la BD,

- soit comme argument de recherche d'une routine d'adressage qui convertit cette clé en adresse disque.

(Les deux méthodes offertes par DL1 pour entrer dans une BD)

Si on a choisi d'avoir un index, cette clé est obligatoirement de type unique (U). Ceci signifie qu'une valeur donnée de cette clé ne peut apparaitre que dans une seule occurrence. Ceci est mentionnée derrière le mot SEQ ou par défaut. Pour un segment dépendant, le champ clé n'est pas obligatoire, bien que sa présence peut être extrêmement précieuse. S'il est défini, DL1 ordonne par valeurs croissantes les occurrences qui dépendent d'un meme parent (=jumeaux physiques).

La présence d'un champ clé à tous les niveaux d'un chemin offre un avantage important.

2. indiquer à DL1 les modalités de réalisation physique de la BD.

La première macro-instruction indique le type d'organisation choisi. La seconde ('DATASET') permet de fournir toutes les indications sur le support physique.

```
                                        HISAM
                                        HSAM
DBD     NAME = DBD1   , ACCES =         HIDAM
                                        HDAM

DATASET     DD1     =
            DEVICE  =
            BLOCK   =
            RECORD  =


    SEGM  NAME = ID                                BYTES = 250
    FIELD NAME = (NOM,SEQ,U)  ,START =   1  ,BYTES =   45
    FIELD NAME = ADRESSE      ,START = 125  ,BYTES =  100
    FIELD NAME = VILLE        ,START = 200  ,BYTES =   25
    SEGM  NAME = CHEQUE ,PARENT = ID       ,BYTES =  60
    FIELD NAME = (NUM,SEQ,U)  ,START =  20  ,BYTES =   12
    SEGM  NAME = MOUV    ,PARENT = CHEQUE  ,BYTES =  40
    FIELD NAME = (DATE,SEQ,U),START =  15  ,BYTES =    5
    FIELD NAME = INDIC        ,START =   1  ,BYTES =    1
    FIELD NAME = SOLDE        ,START =  20  ,BYTES =   20
    SEGM  NAME = TITRE   ,PARENT = ID      ,BYTES = 120
    SEGM  NAME = PRET    ,PARENT = ID      ,BYTES =  60
    SEGM  NAME = REMB    ,PARENT = PRET    ,BYTES =  40


DBDGEN
```

Le projet d'exploitation

Un programme ne peut pas en principe connaitre et exploiter toutes les BD's disponibles. Aussi, un autre jeu de macro-instructions va permettre au responsable des BD's de controler rigoureusement les conditions dans lesquelles un programme va travailler. Il s'agit de sélectionner les segments de ces bases qui seront connus par ce programme, de sélectionner les fonctions qu'il peut faire par DL1 pour chacun de ces segments. Le résultat ce cette opération est le PSB.

1. sélection des BD's connues du programme d'application

Un certain nombre de macro-instructions PCB autorise chacune un accès à une BD qui peut être limité à certains de ses segments types. Ceci est opéré par les macro-instructions SENSEG (suivant la macro-instruction PCB).

2. détermination des fonctions autorisées

Elle s'effectue par le paramètre PROCOPT.

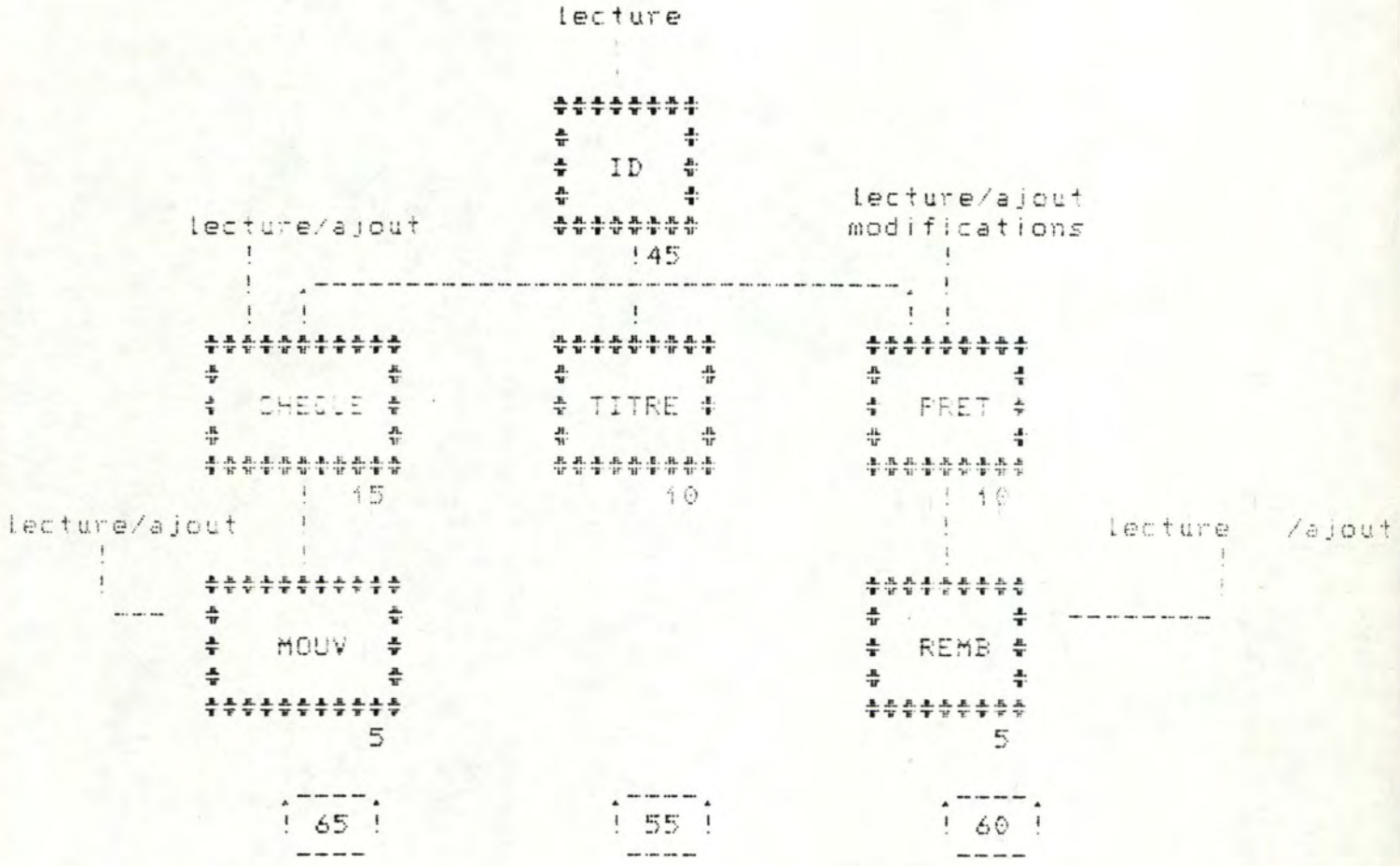


Fig. IMC.6

La macro-instruction PSBGEN indique le nom du PSB et le langage utilisé pour écrire le programme d'exploitation des bases de donnees

```
PCB TYPE = DB, NAME = DBD1
SENSEG   NAME = ID,PROCOPT = G
SENSEG   NAME = CHEQUE,PARENT = ID, PROCOPPT  = GI
SENSEG   NAME = MOUV, PARENT = CHEQUE, PROCOPT  = GI
SENSEG   NAME = PRET,PARENT = ID, PROCOPT  = GIR
SENSEG   NAME = REMB,PARENT = PRET, PROCOPT  = GI

PSBGEN NAME = PSB1, LANG =  ASSEM
                            PL/I
                            COBOL
```

Quand un paramètre PROCOPT est placé dans la macro-instruction PCB il concerne tous les segments référencés, sauf ceux dont la macro-instruction SENSEG contient elle-même un paramètre PROCOPT.

LES BASES DE DONNEES LOGIQUES

Pour illustrer la description de cette notion, nous allons reprendre le même exemple, mais construit sur deux bases de données. Cette décomposition en deux bases de données n'est pas une solution nécessairement performante, mais cela n'est point notre but.

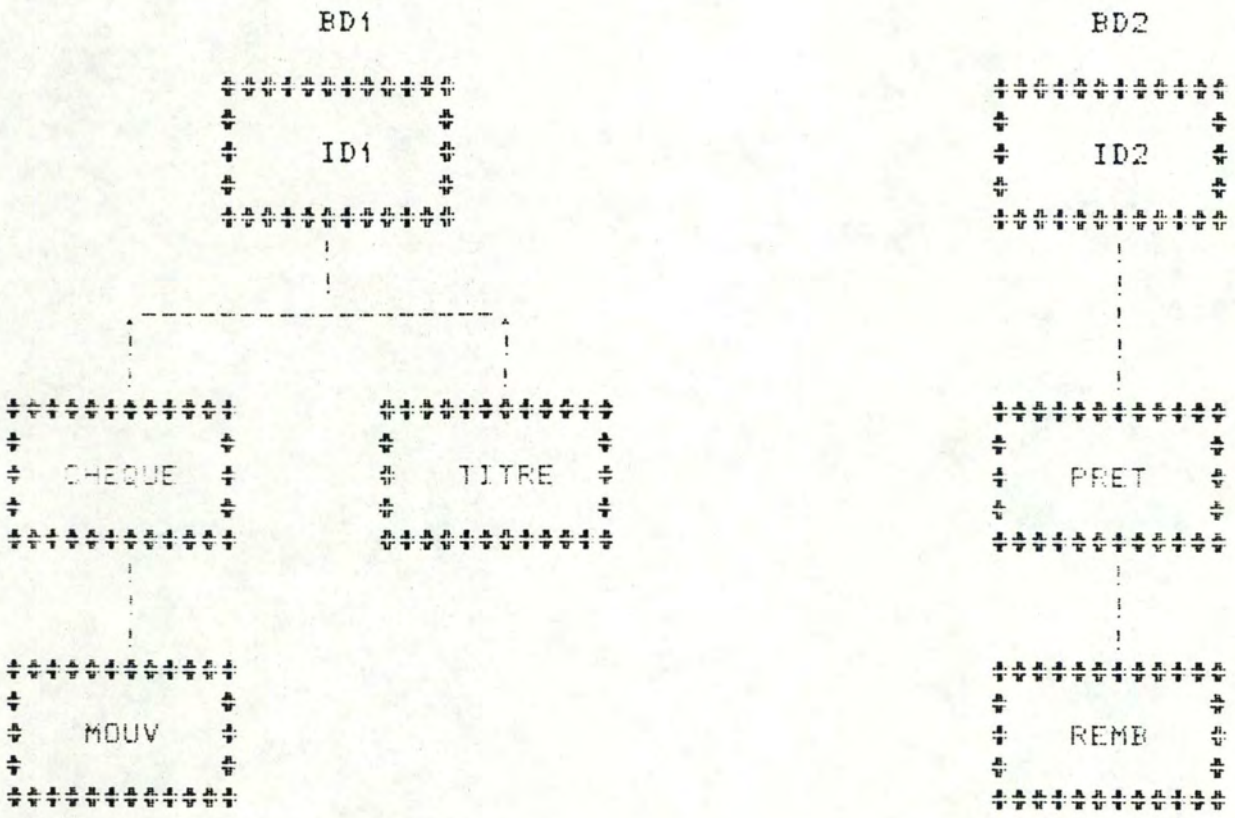


Fig. IMS.7

Une base de données logique, comme une base de donnée physique, est définie au moyen d'un Data Base Description. Ce DBD est dénommé DBD logique par opposition au DBD physique. Chaque DBD logique est défini en termes d'un ou plusieurs DBD's physiques qui doivent toujours exister. Dans notre exemple, nous requérons :

a) un DBD physique pour la BD1,

b) un DBD physique pour la BD2,

c) un DBD logique pour la base de données logique qui couvre les deux premières.

La seule chose à noter est que la macro-instruction SEGM pour le segment ID1 de BD1 est suivie d'une macro-instruction LCHILD spécifiant que le segment PRET de BD2 est un enfant logique de ID1 dans BD1. Dans le second DBD physique, il faut indiquer la relation inverse, c'est-à-dire qu'ID1 dans BD1 est le parent logique de PRET .

Dans le DBD logique, la macro-instruction comprend une clause qui spécifie que l'accès est logique (ACCES = LOGICAL). La macro-instruction DATASET spécifie que le dataset est logique (DATASET LOGICAL). Pour le segment racine, on spécifie la base de données physique où il se trouve. Les macro-instructions FIELD ne doivent pas y etre incluses.

Pour notre exemple, les trois DBD's se présentent ainsi :

```
                                    HISAM
                                    HSAM
DBD     NAME = BD1    , ACCES =     HIDAM
                                    HDAM

DATASET     DD1     =
            DEVICE  =
            BLOCK   =
            RECORD  =


   SEGM  NAME = ID                          BYTES = 250
   LCHILD NAME = (PRET,BD2)
   FIELD NAME = (NOM,SEQ,U) ,START =   1  ,BYTES =   40
   FIELD NAME = ADRESSE     ,START = 125  ,BYTES =  100
   FIELD NAME = VILLE       ,START = 200  ,BYTES =   25
   SEGM  NAME = CHEQUE ,PARENT = ID      ,BYTES =  60
   FIELD NAME = (NUM,SEQ,U) ,START =  20  ,BYTES =   12
   SEGM  NAME = MOUV   ,PARENT = CHEQUE  ,BYTES =  40
   FIELD NAME = (DATE,SEQ,U),START =  15  ,BYTES =   5
   FIELD NAME = INDIC       ,START =   1  ,BYTES =   1
   FIELD NAME = SOLDE       ,START =  20  ,BYTES =  20
   SEGM  NAME = TITRE  ,PARENT = ID      ,BYTES = 120

DBDGEN
```

```
                                    HISAM
                                    HSAM
DBD     NAME = BD2    , ACCES =     HIDAM
                                    HDAM

DATASET     DD1     =
            DEVICE  =
            BLOCK   =
            RECORD  =


   SEGM  NAME = ID2                         BYTES = 250
   SEGM  NAME = PRET
         PARENT=((ID2),(ID1,VIRTUAL,BD2))   ,BYTES =  60
   SEGM  NAME = REMB   ,PARENT = PRET    ,BYTES =  40

DBDGEN
```

```
DBD     NAME = LDB    , ACCES =      LOGICAL

DATASET     LOGICAL


     SEGM   NAME = ID1, SOURCE=((ID1,,BD1))
     SEGM   NAME = CHEQUE ,PARENT = ID1
            SOURCE=((CHEQUE,,BD1))
     SEGM   NAME = MOUV    ,PARENT = CHEQUE ,BYTES =  40
            SOURCE=((MOUV,,BD1))
     SEGM   NAME = TITRE   ,PARENT = ID      ,BYTES = 120
            SOURCE=((TITRE,,BD1))
     SEGM   NAME = PRET    ,PARENT = ID      ,BYTES =  60
            SOURCE=((PRET,,BD2))
     SEGM   NAME = REMB    ,PARENT = PRET    ,BYTES =  40
            SOURCE=((REMB,,BD2))
DBDGEN
```

La structure de la base de données logique est celle-ci :

Fig. IMS.8

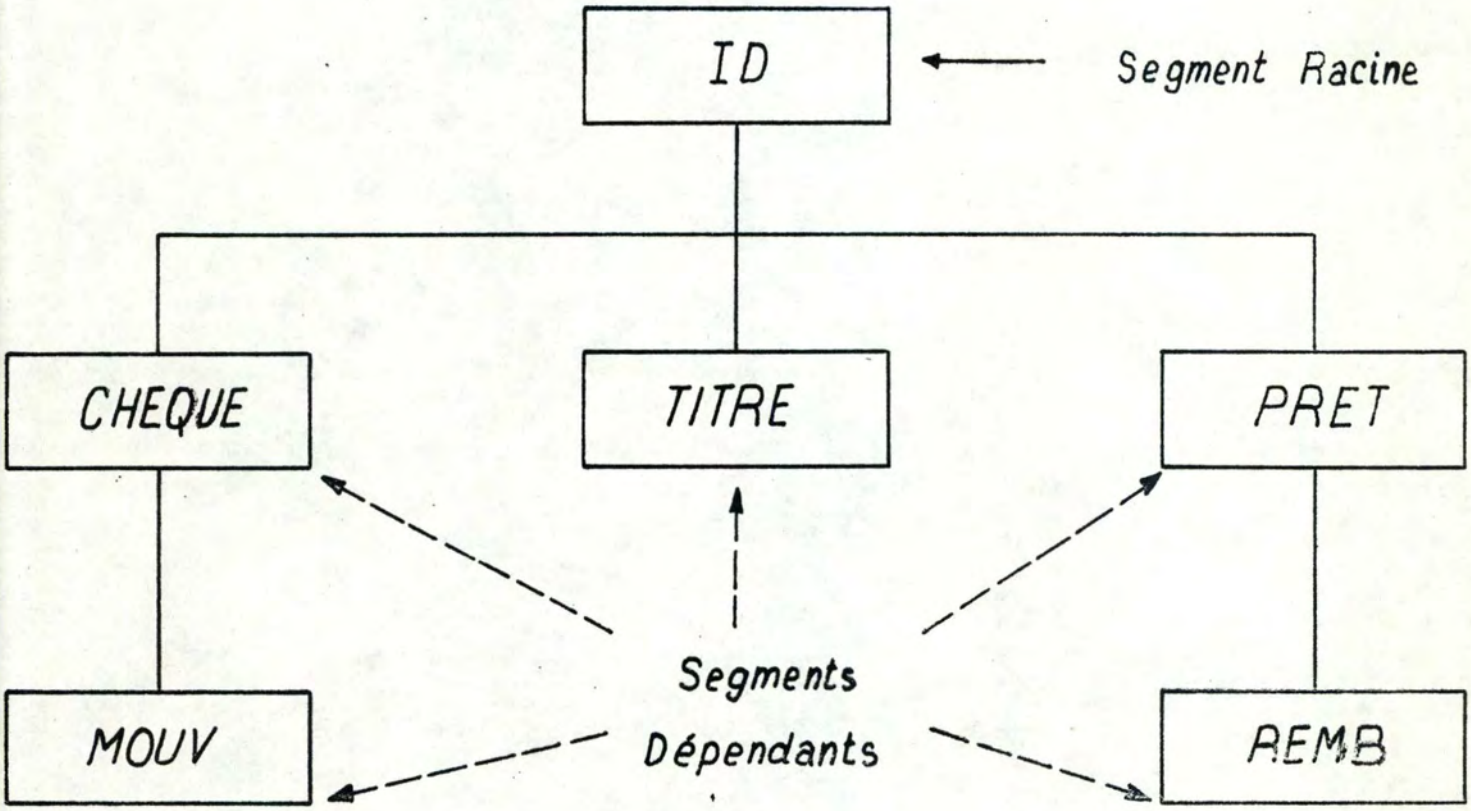


Fig. IMS.1 : La structure arborescente

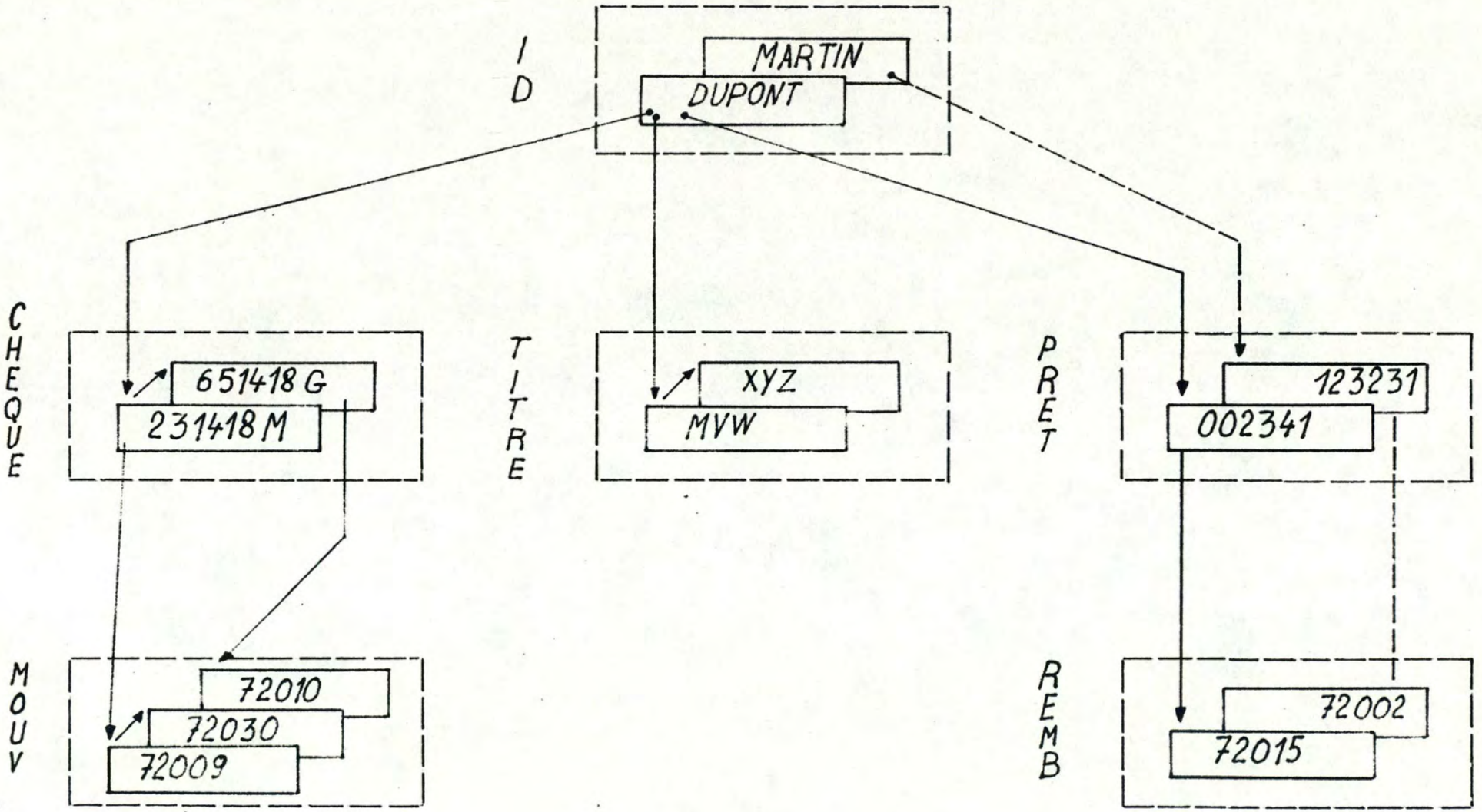

ID
MARTIN
DUPONT
CHEQUE
651418 G
231418 M
TITRE
XYZ
MVW
PRET
123231
002341
MOUV
72010
72030
72009
REMB
72002
72015

Fig. INS.2

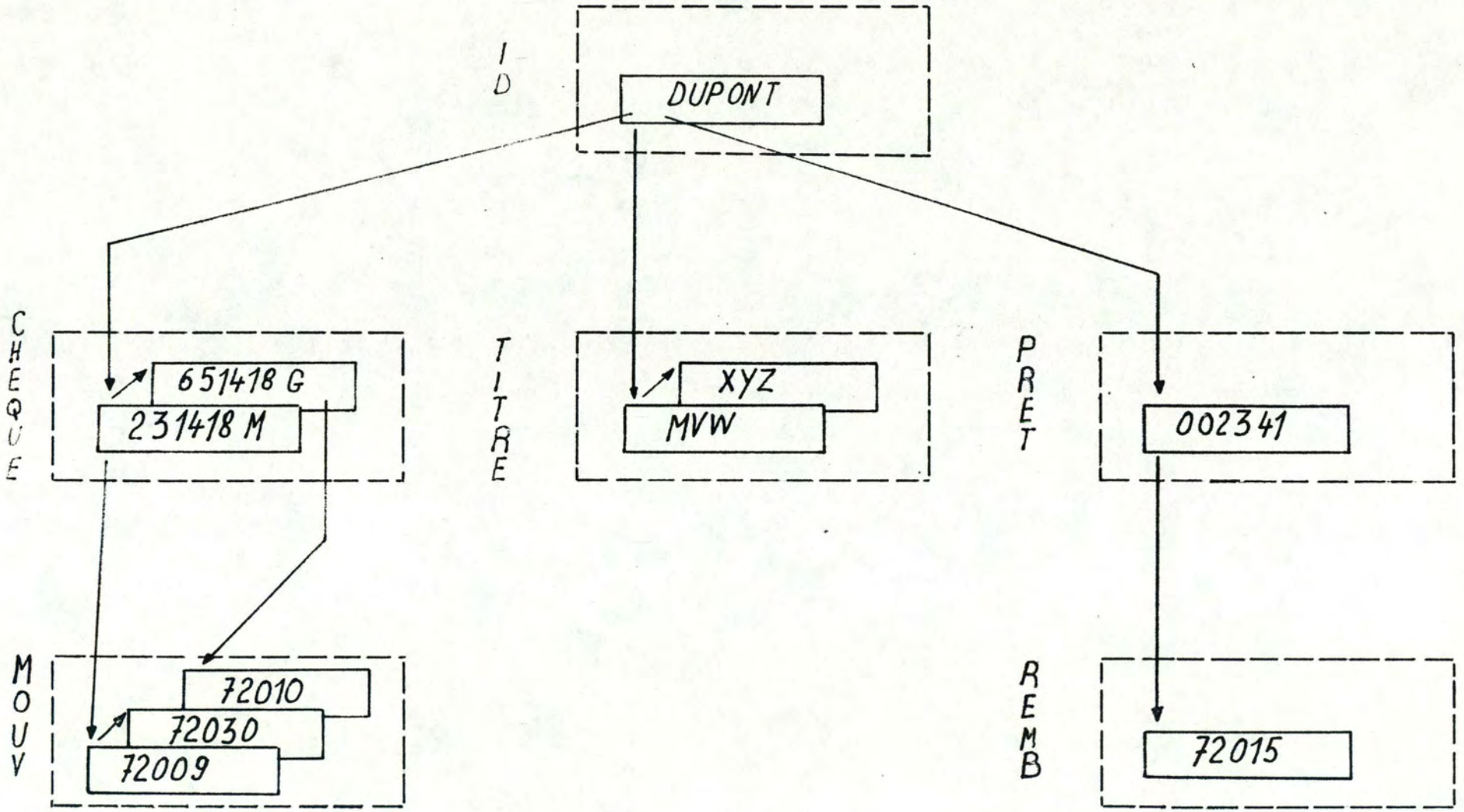


Fig. IMS. 3 : Le "Data Base Record" DUPONT.

Annexe au chapitre de l'analyseur

L'analyse lexicale.

L'analyse lexicale consiste à reconstituer à partir de la suite de caractères une suite de symboles de base.

Nous profitons de cette analyse pour donner à chacun des symboles de base une représentation interne et pour attribuer à chacune de ces représentations internes un nom plus parlant.

En effet, chaque représentation interne prend une valeur hexadécimale codée sur un caractère et son nom est construit comme suit. Chaque nom est préfixé des caractères 'sv_', suivi de quatre lettres qui l'identifie des autres.

Ce procédé permet de codifier le type du symbole de façon très précise tout en autorisant facilement des modifications sur la représentation externe des symboles de base.

L'analyse syntaxique

Toute variable dynamique débute par un champ 'TCONSTR', qui codé sur un caractère hexadécimal identifie le type de construction qui suit. Voici un tableau récapitulant toutes les correspondances.

TABLEAU DE CORRESPONDANCE

TCONSTR	TYPE DE FONCTION
tc_prog	programme
tc_func	déclaration de fonction
tc_proc	déclaration de procédure
tc_apel	appel de fonction ou de procédure
tc_siva	expression de désignation d'une variable simple
tc_vain	expression de désignation d'une variable indicée
tc_redb	expression de désignation d'un composant d'un record ou d'un segment (champ) d'une BD
tc_unop	expression à un opérateur et à un opérande
tc_biop	expression à deux opérandes (et un opérateur binaire)
tc_csre	expression d'une constante réelle
tc_csen	expression d'une constante entière
tc_csca	expression d'une constante chaine de caractères
tc_csbo	expression d'une constante booléenne
tc_afec	instruction d'affectation
tc_if1	instruction conditionnelle premier type
tc_if2	instruction conditionnelle second type
tc_whil	instruction répétitive WHILE
tc_unti	instruction répétitive UNTIL
tc_for	instruction répétitive FOR
tc_inco	instruction composée
tc_etiq	instruction étiquette
tc_goto	instruction de saut
tc_retu	instruction de retour à la fonction appelante
tc_foea	instruction d'accès BD FOR_EACH
tc_find	instruction d'accès BD FIND
tc_flex	instruction d'accès BD FIND suivi de la clause NEXT
tc_grou	instruction d'accès BD GROUPED

TABLEAU DE CORRESPONDANCE

TCONSTR	TYPE DE FONCTION
tc_nezo	instruction d'activation d'une zone
tc_enzo	instruction de désactivation d'une zone
tc_afzo	instruction d'affectation à une zone
tc_vasi	déclaration de variables simples
tc_tab	déclaration de tableaux
tc_indi	déclaration des bornes minimale et maximale d'un indice.
tc_reco	déclaration de variable structurée de type record

Ce second tableau établit la correspondance entre la codification des opérateurs et leur signification. A nouveau, cette codification se fait sur un caractère hexadécimal.

TABLEAU DE CORRESPONDANCE DES OPERATIONS

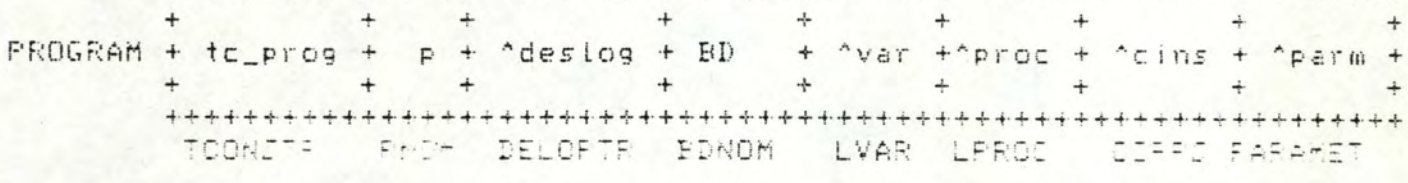
OPER	TYPE D'OPERATION	symbole PL/I
op_plus	addition	+
op_moin	soustraction	-
op_fois	multiplication	*
op_divi	division	/
op_expo	exponentiation	**
	Opérations de comparaison :	
op_infe	inférieur	<
op_supe	supérieur	>
op_ineg	inférieur ou égal	<=
op_sueg	supérieur ou égal	>=
op_egal	égal	=
op_nega	different	^=
op_nsup	pas plus grand	^>
op_ninf	pas plus petit	^<
	Opérations logiques :	**Symbole PL/I**
op_or	ou logique	|
op_and	et logique	&
op_not	non logique	^
	Opérations sur les chaines de caractères :	**Symbole PL/I**
op_conc	concaténation de chaines de caractères	||

1. Représentation interne des déclarations de programme

Soit la déclaration de programme

```
p : PROCEDURE;
decrap;
decdb;
decv1;...;decvN;
decp1;...;decpS;
cins;
```

où : p identificateur du programme
decrap déclaration de la définition logique du rapport,
decdb déclaration de la BD,
decv1;...;decvN des déclarations de listes de variables
decp1;...;decpS des déclarations de fonction ou de procédure,
cins est une instruction composée.



dont voici une déclaration :

```
1 program              BASED,
  2 tconstr            CHAR(1) INIT(tc_prog),
  2 pnom               CHAR(12),
  2 deloptr            PTR INIT(null),
  2 bdnom              CHAR(8),
  2 lvar               PTR INIT(null),
  2 lproc              PTR INIT(null),
  2 corps              PTR INIT(null),
  2 paramet            PTR INIT(null),

progptr                PTR,
```

où :

progptr est un pointeur vers la représentation interne d'un programme LEDGR.
tconstr détermine le type de construction,
pnom l'identificateur du programme,
deloptr est un pointeur vers la représentation interne de la description logique de la définition de rapport,
bdnom est l'identificateur de la BD accédée,
lvar est un pointeur vers la représentation interne de la liste des déclarations de variables,
lproc est un pointeur vers la représentation interne de la liste des déclarations de procédures et fonctions de la définition du rapport,
corps est un pointeur vers la représentation interne de la suite des instructions,
paramet est un pointeur vers la représentation interne de la suite des variables paramètres du rapport.

2. Représentation interne des déclarations de fonction et de procédure

Soit la déclaration de procédure

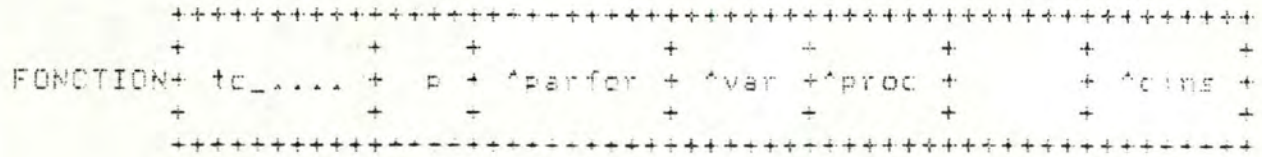
```
PROCEDURE p (parfor1,...,parforN);
decv1;...;decvN;
decp1;...;decpS
cins;
```

où : p identificateur de la procédure,
parfor1,...,parforN des déclarations de paramètres formels,
decv1;...;decvN des déclarations de listes de variables,
decp1,...,decpS des déclarations de fonction ou de procédure,
cins est une instruction composée.

Soit la déclaration de fonction

```
PROCEDURE p (parfor1,...,parforN) : typef;
decv1;...;decvN;
decp1;...;decpS
cins;
```

où : p identificateur de la fonction,
parfor1,...,parforN des déclarations de paramètres formels,
typef le type de la fonction,
decv1;...;decvN des déclarations de listes de variables,
decp1,...,decpS des déclarations de fonction ou de procédure,
cins est une instruction composée.



dont voici une déclaration :

```
1 fonction              BASED,
  2 tconstr             CHAR(1),
  2 pnom                CHAR(12),
  2 lpar                PTR INIT(null),
  2 lvar                PTR INIT(null),
  2 lproc               PTR INIT(null),
  2 ftype,
    3 typefct8,
      4 typefct(8)      BIT,
    3 precision         BIN FIXED(15),
    3 chlapvirg         BIN FIXED(15),
  2 corps               PTR INIT(null),
```

où :

tconstr détermine le type de construction,
pnom l'identificateur de la fonction (procédure),
lpar est un pointeur vers la représentation interne de la suite des déclarations de paramètres formels,
ftype est une codification du type de la fonction,
lvar est un pointeur vers la représentation interne de la liste des déclarations de variables,
lproc est un pointeur vers la représentation interne de la liste des déclarations de procédures et fonctions propres à celle-là,

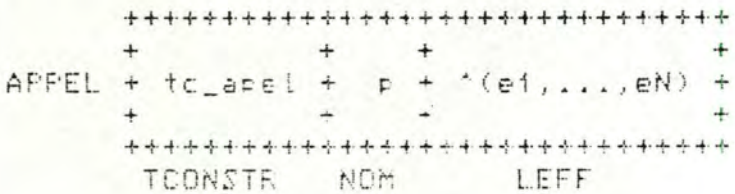
corps est un pointeur vers la représentation interne de la suite des instructions.

Les déclarations de procédure sont représentées de la meme maniere, sauf pour la valeur de TCONSTR qui sera tc_proc et celle de FTYPE qui sera indéfinie (quelconque).

Dans le corps des procédures et fonctions, nous interdisons l'emploi des instructions d'accès aux DBs, ainsi que celles de génération. Notre reflexion a ce sujet nous a permis de nous rendre compte que cet usage soulèverait de nombreux problèmes non triviaux (passage des paramètres, quel(s) PCB(s) utiliser pour exécuter cette procédure (fonction),...) qui ne pourraient etre analysés et résolus dans le cadre de ce mémoire sans négliger le reste de la solution. Nous nous contentons de soulever un voile sur ces problèmes en laissant la porte grande ouverte à une étude ultérieure.

3. Représentation des appels de fonctions et de procedures

L'appel de fonction ou de procédure p(e1,...,eN)
est représenté par une variable dynamique "APPEL" dont voici une illustration :



En voici une déclaration, également :

```
DCL 1 appel                BASED,
      2 tconstr            CHAR(1),
      2 nom                CHAR(12),
      2 leff               POINTER INIT(null);
```

où :

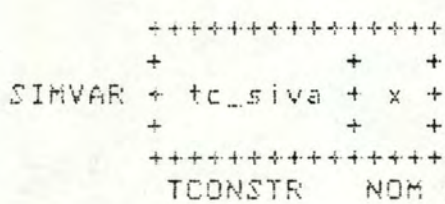
tconstr détermine le type de construction,
nom l'identificateur de la fonction (procédure) appelée,
leff est un pointeur vers la représentation interne de la liste des paramètres effectifs.

Il est à rappeler également qu'un appel de procédure peut se faire également de la façon suivante :

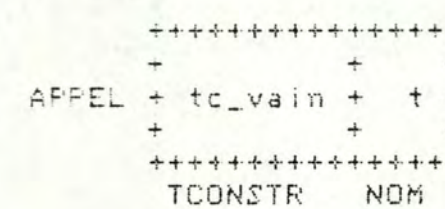
CALL p(e1,e2,...,eN)

4. Représentation interne des expressions :

Les expressions de désignation x , t[i] et t[i,j]
où x et t sont des identificateurs et i et j des expressions arithmétiques, se représentent par des variables dynamiques qui s'appellent respectivement "SIMVAR" et "APPEL" dont voici des illustrations :



et



En voici des déclarations, également :

```
DCL 1 simvar                BASED,
      2 tconstr             CHAR(1) INIT('tc_siva'),
      2 nom                 CHAR(12),
      2 simptr              POINTER INIT(null);
```

où :

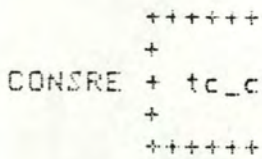
tconstr détermine le type de construction,
nom l'identificateur de la variable désignée,
simptr est un pointeur vers la représentation interne, du reste de l'expression de désignation.

```
DCL 1 appel                 BASED,
      2 tconstr             CHAR(1),
      2 nom                 CHAR(12),
      2 leff                POINTER INIT(null);
```

où :

tconstr détermine le type de construction,
nom l'identificateur du tableau appelé,
leff est un pointeur vers la représentation interne des indices effectifs.

Les constantes sont représentées :

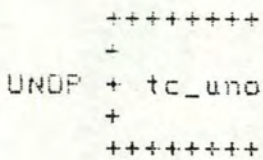



```
DCL 1 consre                BASED,
      2 tconstr             CHAR(1) INIT(tc_csre),
      2 consval             BIN FLOAT(109),

DCL 1 consent               BASED,
      2 tconstr             CHAR(1) INIT(tc_csen),
      2 consval             DEC FIXED(15),

DCL 1 conscar BASED,
      2 tconstr             CHAR(1) INIT(tc_csca),
      2 consval             CHAR(150) VARYING;
```

Les constantes chaines de caractères doivent etre encadrées par un des deux symboles quote (') ou double quote ("). Si le symbole delimiteur doit se retrouver dans la constante, chacune de ces occurrences devra etre doublée au sein de celle-là.

```
DCL 1 consbool,
      2 tconstr             CHAR(1) INIT(tc_csbo),
      2 consval             BIT(1),
```

Les expressions de la forme w e1
ou e1 est une expression et,
w un opérateur unaire.



où w' est la représentation de w par une valeur de type character.

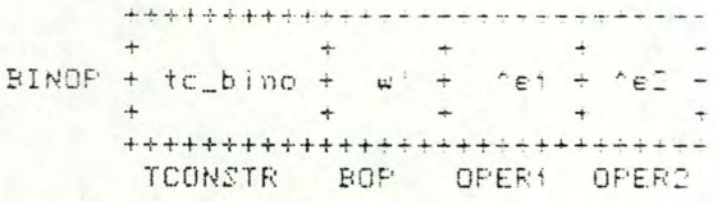
```
DCL 1 unop                  BASED,
      2 tconstr             CHAR(1) INIT(tc_unop),
      2 uop                 CHAR(1),
      2 oper                POINTER INIT(null),
```

où :

tconstr détermine le type de construction,
uop une codification de l'opérateur,
oper est un pointeur vers la représentation interne de l'expression sur laquelle porte l'opérateur.

Soit les expressions de la forme e' = e2
où w = opérateur binaire,
et e1,e2 expressions dont la forme dépend de w selon les règles de syntaxe et de sémantique du LEDGR.

Leur représentation sera du type :




```
DCL 1 binop                BASED,
      2 tconstr            CHAR(1) INIT(tc_bino),
      2 bop                CHAR(1),
      2 oper1              POINTER INIT(null),
      2 oper2              POINTER INIT(null);
```

où :

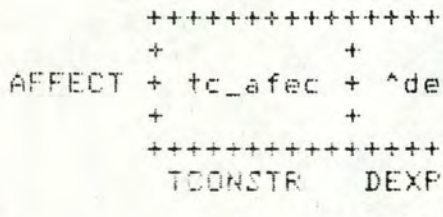
tconstr détermine le type de construction,
bop une codification de l'opérateur binaire
oper1 est un pointeur vers la représentation interne de la première expression sur laquelle porte l'opérateur,
oper2 est un pointeur vers la représentation interne de la seconde expression sur laquelle porte l'opérateur.

5. Représentation interne des instructions :

5.1 instruction d'affectation (assignation):

de := expr

où : de symbolise une expression de désignation,
expr symbolise une expression,
se représente :




```
DCL 1 affect              BASED,
      2 tconstr           CHAR(1) INIT(tc_afec),
      2 dexpr             POINTER INIT(null),
      2 expr              POINTER INIT(null);
```

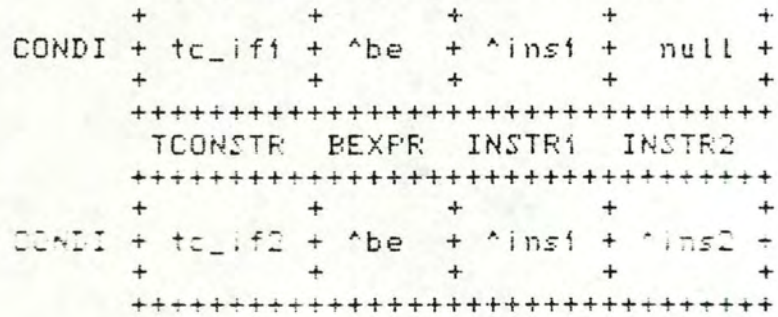
où :

tconstr détermine le type de construction,
dexpr est un pointeur vers la représentation interne de l'expression de désignation sur laquelle va porter l'affectation,
expr est un pointeur vers la représentation interne de l'expression qui déterminera la valeur à affecter à la variable désignée.

5.2 instructions conditionnelles :

IF be THEN ins1

IF be THEN ins1 ELSE ins2 se représentent :



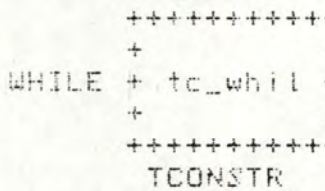
```
DCL 1 condi                BASED,
      2 tconstr            CHAR(1),
      2 bexpr              POINTER INIT(null),
      2 instr1             POINTER INIT(null),
      2 instr2             POINTER INIT(null);
```

où :
tconstr détermine le type de construction,
bexpr est un pointeur vers la représentation interne de l'expression booléenne qui détermine le choix à effectuer,
ins1 est un pointeur vers la représentation interne de l'instruction à exécuter dans le cas d'un résultat affirmatif à la condition,
ins2 est un pointeur vers la représentation interne de l'instruction à exécuter dans le cas d'un résultat négatif à la condition.

5.3 Instructions répétitives :

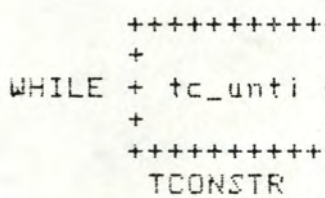
WHILE (be) ins;

où : ins symbolise une instruction simple ou composée, se représente :



UNTIL (be) ins;

où : ins symbolise une instruction simple ou composée, se représente :

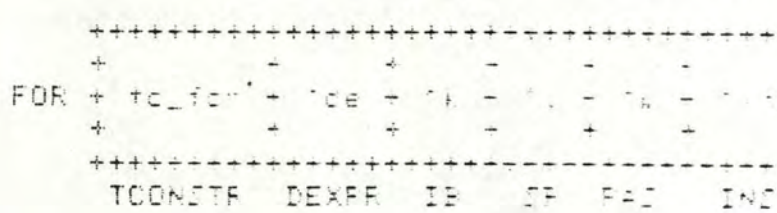



```
DCL 1 while                BASED,
      2 tconstr            CHAR(1),
      2 bexpr              POINTER INIT(null),
      2 instr              POINTER INIT(null);
```

ou :
tconstr détermine le type de construction,
bexpr est un pointeur vers la représentation interne de l'expression booléenne qui détermine la fin de la répétition,
instr est un pointeur vers la représentation interne de l'instruction à répéter.

FOR de := expr1 TO expr2 BY expr3; ins;

ou : ins symbolise une instruction simple ou composée,
expr1, expr2, expr3 symbolisent des expressions.
se représente :

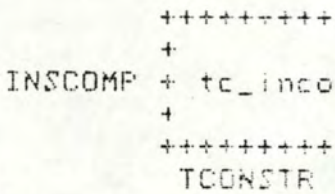



```
DCL 1 for                    BASED,
      2 tconstr              CHAR(1) INIT(tc_for),
      2 dexpr                POINTER INIT(null),
      2 bi                   POINTER INIT(null),
      2 bs                   POINTER INIT(null),
      2 pas                  POINTER INIT(null),
      2 instr                POINTER INIT(null),
```

ou :
tconstr détermine le type de construction,
dexpr est un pointeur vers la représentation interne de l'expression de désignation de la variable compteur de la répétition,
bi est un pointeur vers la représentation interne de l'expression de la borne inférieure de la variable compteur,
bs est un pointeur vers la représentation interne de l'expression de la borne supérieure de la variable compteur,
pas est un pointeur vers la représentation interne de l'expression du pas d'avancement de la variable compteur,
instr est un pointeur vers la représentation interne de l'instruction à répéter.

5.4 instruction composée :

DO; lins; END;
où : lins symbolise une liste d'instructions,
se représente :

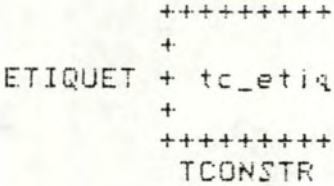



```
DCL 1 inscomp              BASED,
      2 tconstr            CHAR(1) INIT(tc_inco),
      2 linstr             POINTER INIT(null);
```

où :
tconstr détermine le type de construction,
dexpr est un pointeur vers la représentation interne de l'expression de désignation de la variable compteur de la répétition,
linstr est un pointeur vers la représentation interne de la suite d'instructions à exécuter.

5.5 instruction étiquette :

ident : instr;
où : ident symbolise un identificateur,
instr symbolise une instruction quelconque,
se représente :

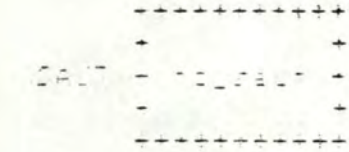



```
DCL 1 etiquet              BASED,
      2 tconstr            CHAR(1) INIT(tc_etiq),
      2 nometiq            CHAR(10);
```

où :
tconstr détermine le type de construction,
nometiq désigne l'identificateur de l'instruction d'étiquette.

5.6 instruction de saut :

GOTO ident;
où : ident symbolise un identificateur,
se représente :



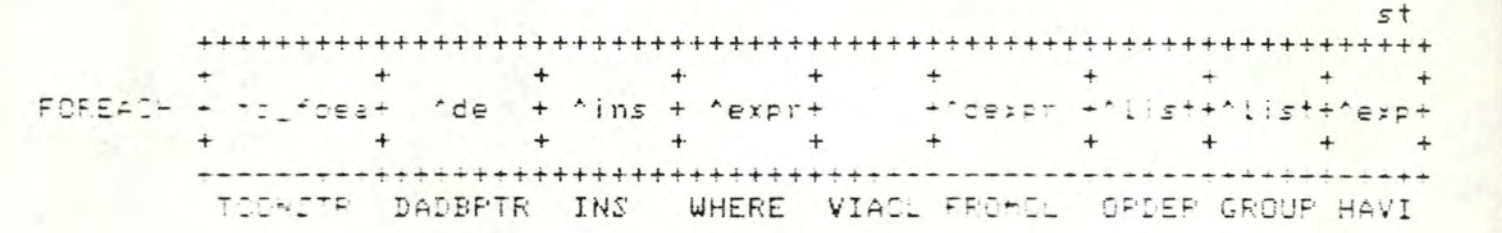

```
DCL 1 saut BASED,
      2 tconstr CHAR(1) INIT(tc_saut),
      2 nomsaut CHAR(12);
```

où :
tconstr détermine le type de construction,
nomsaut désigne l'identificateur de l'étiquette vers laquelle se fera le saut.

5.7 Les instructions d'accès aux BD's

5.7.1 L'instruction for_each

FOR_EACH de clauses ins;
où : de symbolise une expression de désignation d'un segment, champ d'une BD,
clauses symbolise un ensemble de clauses spécifiques à cette instruction for_each,
ins symbolise une instruction quelconque.



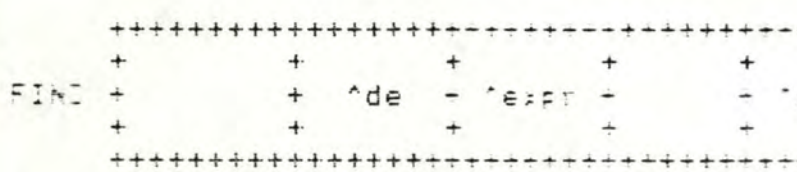
```
DCL 1 foreach              BASED,
      2 tconstr            CHAR(1) INIT(tc_foea),
      2 dadbptr            POINTER,
      2 where              PTR INIT(null),
      2 viacl              CHAR(12),
      2 fromcl             CHAR(12),
      2 order              PTR INIT(null),
      2 grouped            PTR,
      2 havi               PTR,
      2 ins                POINTER;
```

où :

tconstr détermine le type de construction,
dadbptr est un pointeur vers la représentation interne de l'expression de désignation du segment sur lequel la sélection va s'effectuer
ins est un pointeur vers la représentation interne de l'instruction à effectuer sur les segments sélectionnés,
where est un pointeur vers la représentation interne de l'expression logique qui détermine la clause 'where',
viacl est l'identificateur d'un chemin,
fromcl est un identificateur d'une variable d'article
order est un pointeur vers la représentation interne d'une liste de champs lesquels déterminent l'ordre des champs sélectionnés,
order est un pointeur vers la représentation interne d'une liste de champs,
having est un pointeur vers la représentation interne d'une expression.

5.7.2 L'instruction find

FIND de clauses;
où de symbolise une expression de désignation,
clauses symbolise un ensemble de clauses spécifiques à l'instruction,
se représente :



```
DCL 1 find                  BASED,
      2 tconstr             CHAR(1),
      2 dadbptr             POINTER INIT(null),
      2 where               POINTER INIT(null),
      2 fromcl              CHAR(12),
      2 viacl               CHAR(12);
```

où :

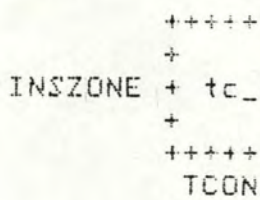
tconstr détermine le type de construction,
dadbptr est un pointeur vers la représentation interne de l'expression de désignation du segment sur lequel la sélection va s'effectuer,
viacl est l'identificateur d'un chemin,
fromcl est un identificateur d'une variable d'extrémité,
where est un pointeur vers la représentation interne de l'expression logique qui détermine la clause "where";

5.8 Les instructions de génération

5.8.1 L'activation et la désactivation

Celles-ci seront représentées par la variable dynamique à deux composantes. La première permet de distinguer ces deux instructions l'une de l'autre par leur valeur de 'tconstr'.
Pour l'activation (ne_zone) tconstr = tc_nezc
Pour la désactivation (end_zone) tconstr = tc_enzo

La deuxième composante donne le nom de la zone visée par l'instruction.



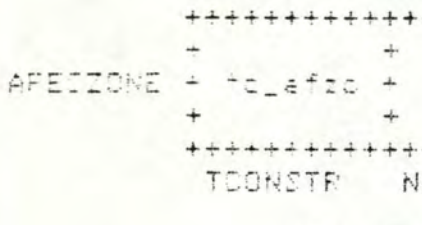

```
DCL 1 inszone               BASED,
      2 tconstr             CHAR(1) INIT(tc_aedt),
      2 nomzone             CHAR(12);
```

où :

tconstr détermine le type de construction,
nomzone détermine l'identificateur de la zone à générer

5.8.2 L'instruction d'affectation à une zone

Cette représentation ressemble fortement à celle de l'instruction classique d'affectation.



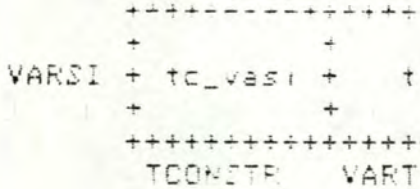

```
DCL 1 afeczone                  BASED,
      2 tconstr                 CHAR(1) INIT(tc_afzo),
      2 nomzone                 CHAR(12),
      2 expr                    PTR;
```

où

tconstr détermine le type de construction
nomzone détermine l'identificateur de la zone sur sur laquelle l'affectation porte,
expr est un pointeur vers la représentation interne d'une expression.

6. Représentation interne des déclarations de variables simples et de tableaux :

La construction x1,...,xN : t pour une déclaration de variables simples se représente sous la forme :



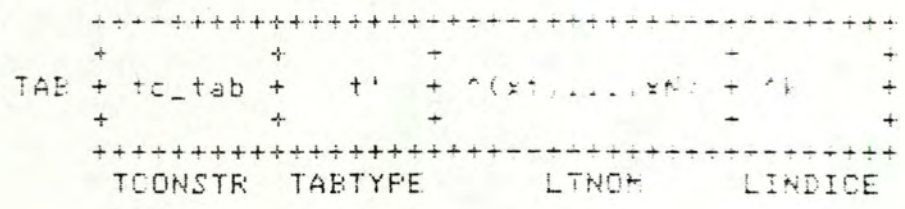

```
DCL 1 varsi                     BASED,
      2 tconstr                 CHAR(1) INIT(tc_vasi),
      2 vartype,
        3 typevar8,
          4 typevar(8)              BIT,
        3 precision                 BIN FIXED(15),
        3 chiapvirg                 BIN FIXED(15),
      2 lvnom                   POINTER;
```

où :

tconstr détermine le type de construction,

vartype est une codification du type de la variable:
typevar détermine une codification du type proprement dit,
precision détermine la precision du type,
chiapvirg détermine le nombre de chiffres après la virgule,
lvnom est un pointeur vers la représentation interne de la liste d'identificateurs des variables de meme type.

Les constructions
t1,...,tN ((déclaration des bornes d'indices)) : t pour une déclaration de tableaux de dimension N
se représentent comme suit :




```
DCL 1 tab                          BASED,
      2 tconstr                    CHAR(1) INIT(tc_tab),
      2 tabtype,
        3 typevar8,
          4 typevar(9)             BIT,
        3 precision                BIN FIXED(15),
        3 chiapvirg                BIN FIXED(15),
      2 ltnom                      POINTER INIT(null),
      2 lindice                    POINTER INIT(null);
```

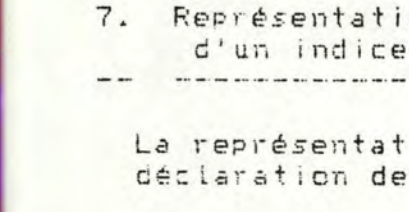
où :
tconstr determine le type de construction,
tabtype
typevar détermine une codification du type proprement dit,
precision détermine la precision du type,
chiapvirg détermine le nombre de chiffres après la virgule,
ltnom est un pointeur vers la représentation interne de la liste d'identificateurs des variables de meme type,
lindice est un pointeur vers la représentation interne de la liste des bornes d'indice de la declaration de tableaux.

7. Représentation des bornes minimale et maximale d'une déclarati on d'un indice de tableau :

La représentation choisie des bornes d'indice permet d'envisager la déclaration de tableaux à N dimensions.

⟨déclaration de bornes d'indices⟩ ::=
⟨intg⟩ : ⟨intg⟩ ! ⟨intg⟩ : ⟨intg⟩,⟨déclaration de bornes d' ind ces

Par défaut, la borne d'indice minimale est 1. Dans ce cas, la borne maximale est déclarée simplement sans les '2 points'.




```
DCL 1 indice                BASED,
      2 tconstr             CHAR(1) INIT(tc_indi),
      2 binf                BIN FIXED(15),
      2 bsup                BIN FIXED(15),
      2 indptr              POINTER INIT(null);
```

où :

tconstr détermine le type de construction,
binf est un entier déterminant la borne inférieure d'un indice,
bsup est un entier déterminant la borne supérieure d'un indice,
indptr est un pointeur vers la déclaration suivante de bornes d'indice du tableau.

8. Représentation de la déclaration d'une variable RECORD :

La structure majeure doit etre déclarée avec le numéro de niveau égale à 1. Les structures mineures et noms élémentaires doivent etre déclarés avec des numéros de niveau arithmétiquement supérieur à 1. Ils doivent etre des constantes entières décimales. Un blanc doit separer les numeros de niveau et le nom associe.

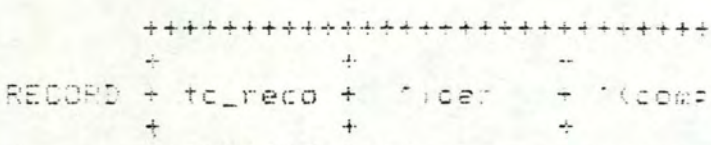
Les numéros choisis pour les niveaux plus internes ne doivent pas etre les entiers successifs immédiats.

Une structure mineure de niveau N contient tous les noms de niveau supérieur à N situés entre le nom de la structure mineure et le nom suivant avec un numéro de niveau inférieur ou égal à N.

Une structure majeure ne peut contenir plus de 15 niveaux.

```
DCL 01 ident,
       ent decl1,
       ent decl2,
          ...
       ent declN;
```

où : ident est l'identificateur de la structure,
decl1,...,declN sont des declarations de variables, composantes de la structure,
ent est un entier superieur representant un numéro de niveau qui doit etre superieur à celui qui le renferme.
se represente :




```
DCL 1 record                BASED,
      2 tconstr             CHAR(1) INIT(tc_reco),
      2 nomrecord           POINTER INIT(null),
      2 listcomp            POINTER INIT(null);
```

où :
tconstr determine le type de construction,
nomrecord est un pointeur vers la représentation interne de la liste d'identificateurs de records de meme structure,
listcomp est un pointeur vers la représentation interne de la liste de composants du record (suite de constructions).

9. Représentation interne des suites de constructions :

Les suites de constructions comportant un nombre quelconque d'éléments sont représentées par une suite de variables dynamiques de type "LCONSTR" dont voici une définition :

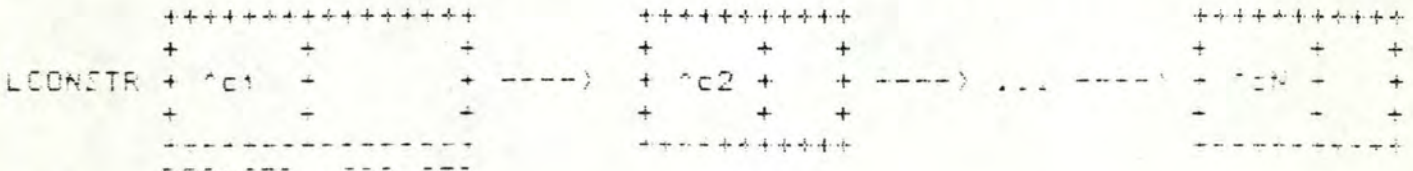
```
DCL 1 lconstr           BASED
      2 pconstr         POINTER INIT(null),
      2 lsconstr        POINTER INIT(null),
```

où :

pconstr est un pointeur vers la représentation interne de la construction qui y correspond,

lsconstr est un pointeur vers la représentation interne du "maillon" suivant de la suite de constructions.

Soit c1,...,cN une suite de constructions du langage LEDGR (expressions, instructions, déclarations d'arguments ou de paramètres formels, de variables ou de tableaux). Cette suite sera représentée par :



Si la suite de constructions est non vide (n >= 1), (^c1 ... ^cN) est le pointeur vers la première cellule de la suite de cellules, sinon c'est le pointeur NULL.

10. Représentation interne des suites d'identificateurs

Les suites d'identificateurs figurant dans les déclarations de variables ou de tableaux et dans les groupes d'arguments et de paramètres formels, d'un programme LEDGR, seront représentées par une suite de variables dynamiques de type "LIDENT" définie par :

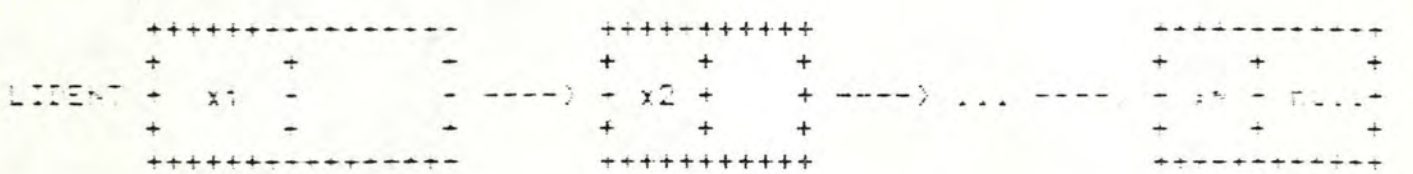
```
DCL 1 lident              BASED,
      2 pident            CHAR(12),
      2 identptr          POINTER INIT(null);
```

où :

pident désigne l'identificateur courant de la suite.

identptr est un pointeur vers la représentation interne du "maillon" suivant de la suite d'identificateurs.

La suite d'identificateurs sera représentée comme suit :



^(x1,x2,...,xN) est le pointeur vers la première cellule.